

BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-93-M14

“3D Texture Synthesis”

by

John J. Krupka

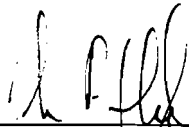
3D Texture Synthesis

John J. Krupka
Department of Computer Science
Brown University

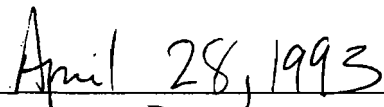
Submitted in partial fulfillment of the requirements for the Degree of
Master of Science in the Department of Computer Science at Brown University

April 1993

This research project by John J. Krupka is accepted in its present form by the
Department of Computer Science at Brown University in partial fulfillment of the requirements
for the Degree of Master of Science.



Professor John F. Hughes
Advisor



Date

3D Texture Synthesis

John J. Krupka
Department of Computer Science
Brown University

May 3, 1993

Contents

1	Introduction	2
2	Research Objectives	2
3	Related Work	3
4	TX : a 3D Texture Synthesizer	3
5	Design Approach	8
5.1	GUI	8
5.2	Genetic Engine	9
5.3	RenderMan Interface	10
5.4	Network Controller	11
5.5	Renderer Independence	11
6	Texture Space	11
6.1	Genetics-Artificial Evolution	11
6.2	Genetic Operations	12
6.3	S-Expression Language	15
6.4	Genetic Parameters	16
6.5	Mapping 3-Space to Color Space	17
6.6	3D Texture Sampling	19
6.7	Getting Started - Something for Nothing	21
7	Results	23
7.1	Examples	23
7.2	Future Work	26
7.3	Conclusions	26
7.4	Acknowledgements	31

Abstract

We present a system for generating 3D textures in an exploratory fashion, using genetic algorithms, distributed rendering, and the RenderMan Shader Language. The 3D textures are generated in the form of RenderMan surface shaders and as such can be applied to any surface shape.

Keywords: 3D textures, solid textures, RenderMan, genetic algorithms.

1 Introduction

A *3D texture* is a function which maps all points in object space to points in a given color space. 3D textures have also been called *solid textures*. More generally, this function is a mapping from one 3 space to another, and can be used to define other object attributes such as opacity, surface normal, surface displacement, specular coefficients, and even density. In this work, we have concentrated on 3D texture functions that determine the color of an object by sampling the function at the object's surface.

3D textures can greatly enhance the realism and interest of computer graphics images and animations. 3D textures have several advantages over 2D textures applied to surface representations. A 3D texture is defined over all of object space so that if an object is cut or carved or sliced one will see the internal texture, not a re-mapping of a 2D texture over the new surface. The visual effect is more interesting in animation, where the shape of an object can be changed by getting cut or gouged or carved, and the object's internal texture structure will get re-displayed.

3D textures are a kind of procedural modeling, and as such, some renderers do not support them. Any renderer which adheres to the RenderMan Interface [Ups85] allows support for 3D textures by use of "shaders". The generation of interesting shaders to date has involved considerable effort (trial and error) and a reasonable level of 3D graphics knowledge. Most are hand crafted with a specific effect in mind.

The set of all possible 3D textures is infinite. One of the challenges of generating interesting 3D textures is in investigating large portions of this set rather than tiny sections of it. Another challenge is to avoid generating textures that are axis-aligned or have noticeable axis behavior. Both of these are quite difficult, and there is a need for a means for a novice to generate 3D textures (or view existing 3D textures) and save them in a re-usable form.

This paper describes a system whose purpose is to let the non-technical user rapidly investigate large regions of texture space, by means of so-called "genetic algorithms": the user starts with some particular texture (or no texture at all) and selectively mutates or combines it with other textures to generate new ones. By keeping around textures that are "interesting" and discarding others from the population, the user can guide the search for interesting textures without ever examining the details of the procedure that created the textures.

2 Research Objectives

The goal of this system was to give the non-technical user a means to generate interesting 3D textures which can then be saved and applied to objects in a still image or an animation. Towards this goal, the research project had the following objectives:

Ease of use: The GUI should be appropriate for a non-technical user. .

Speed: The texture generation/selection iteration cycle should be fast enough that interesting textures are generated before a user gets bored.

Flexibility: The design should be renderer-independent and the textures must be able to be saved in a re-usable form.

Our approach was to combine genetics generation of s-expressions, distributed rendering, a pick-and-choose GUI, a multi-image GUI, and an interface to a RenderMan renderer which supports the RenderMan Shader Language. In order to explain this more fully, we first provide references to the relevant literature, and then describe the program, first from a user's eye view, and then from an more internal view, indicating design decisions with regard to the genetic exploration of 3D texture space.

3 Related Work

The fundamental notions that are used in this work are gathered from a number of ideas presented elsewhere. They are solid texturing, genetic algorithms, and procedural descriptions of textures for generic rendering. (This ignores the engineering issues of design, and the use of distributed processing in doing the rendering work). We therefore give a brief indications of the sources of these notions.

Peachy [Pea85] introduced the notion of solid texturing and created 3D textures of wood grain and marble. Peachy also demonstrated the use of Fourier synthesis in in generating solid textures. At about the same time, Perlin [Per85] introduced the term and concept of solid texture and created some textures which resembled clouds, marble, rock. Perlin also demonstrated the use of noise and turbulence as important components of generating interesting 3D textures.

Perlin [Per89] expanded the use of 3D functions to describe geometry by using a 3D function to determine an objects density.

Sims [Sim91] a variation of the genetic search algorithm was used to generate 2D images, 3D plants and solid textures. This was based on the manipulation of symbolic lisp expressions. For those unfamiliar with the notion of genetic algorithms, we give a brief review and description in Section 6.1.

Upstill [Ups85] describes the RenderMan Shader Language support for the concept of 3D textures. Example surface shaders are described for modeling marble and wood.

4 TX : a 3D Texture Synthesizer

Our 3D texture synthesizer is named "TX," and allows a user to generate 3D textures by simply using the mouse to choose between a few operations. After TX is started it presents the user with a 4x5 matrix of blank image windows along with a menubar and a text input area with some buttons. See Figure 1.

The first thing a user has to do is to specify the host(s) on which the rendering is to done. These hosts are called *rones*. This selection is done by selecting "Drones" from the menubar. The user has two choices: read a list of hosts or start them individually. If the user chooses to specify them individually, he is given two options: naming a

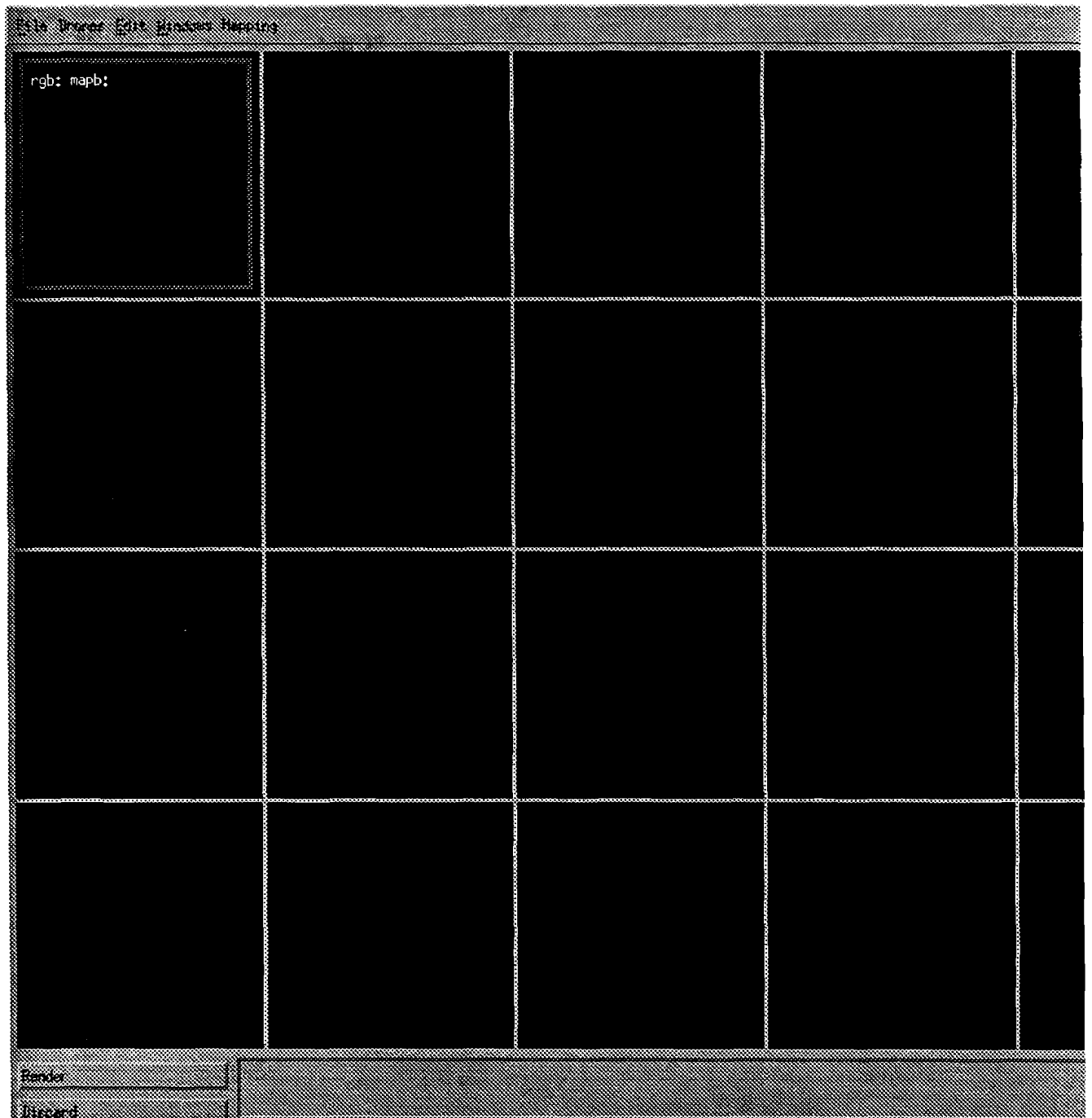


Figure 1: User's Initial View

host explicitly or having the Quahog [Baz93] program pick one for him. If the user chooses to name a drone/host explicitly, a pop-up text widget appears. If the drone is started successfully it will appear in a scrolled list.

The next thing the user has to do is to create a 3D texture function. This can be done by typing one directly, reading a saved one, or creating one randomly. To create one randomly the user selects “Edit” from the menubar. The user then selects “Mutate.”

This will create a number 3D texture functions and send them off to the drones to be rendered. (The number of 3D texture functions generated can be set by selecting “Num Children” from the “Edit” pulldown.) Each texture function is associated with a specific image window. If the user wants to display the s-expression associated with a window, the user selects that window with the left mouse button, and the s-expression gets displayed in the text field at the bottom (see Figure 2).

When each drone is finished rendering a texture, it sends the image back to the GUI and the image is displayed in an image window. Note that the user does not have to wait for any or all of the images to be rendered before interacting further or initiating other operations from the GUI.

Once some or all of the rendered images are displayed, it is the time for the user to judge the fitness of the associated 3D texture functions. If the user is satisfied with an image, he can save the s-expression. If the user is not satisfied with an image, he can either clear it or modify it. If the user want to modify (propagate) a function, he can do so in any of three ways: mutating, crossing, and blending. *Mutation* involves the modification of a single s-expression (function), while crossing and blending require two. To mutate the user must select a window with the left mouse button. This puts a red border around the image and this defines the current window.

To cross or blend the user must select two “parent” images by pressing the middle mouse button. This puts a green border around the image windows. The user then selects “Cross” or “Blend” from the the “Edit” menu (see Figure 3). Selecting an image window with the right mouse button holds/saves the image, which allows the user to clear all the “unheld” windows. This “holding” of an image is indicated by a purple border around the image window. Selecting the image again with the right button “unholds” the image and removes the purple border.

In each image window is a text string like “rgb mapa”. This indicates the final mapping of the 3D texture function into color space. The user can select a mapping for an individual window by selecting “Mapping” from the menubar. The user can select from *color* or *space* mapping. The space mapping maps the result of the 3D texture function into a space limited to 0.0 to 1.0. Color mapping maps this limited 3-space into a color space. These mappings must be done BEFORE the 3D texture function is rendered. The choice of these two mapping functions can greatly change the appearance of an image,

After a few iterations of mutating, blending, crossing, and clearing, the user can save the s-expressions for later input. The user can also save the 3D texture as a

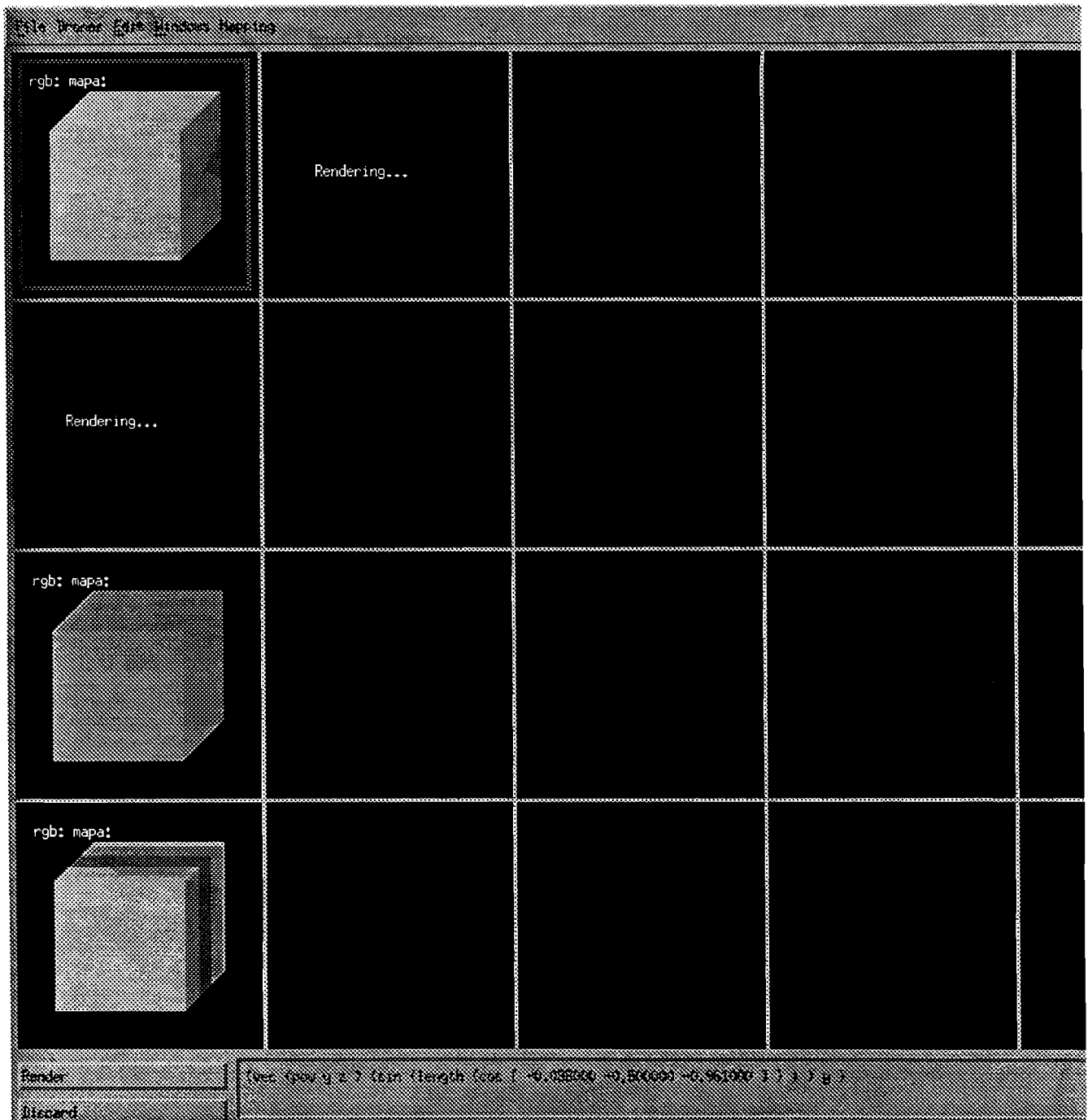


Figure 2: User's Initial Textures

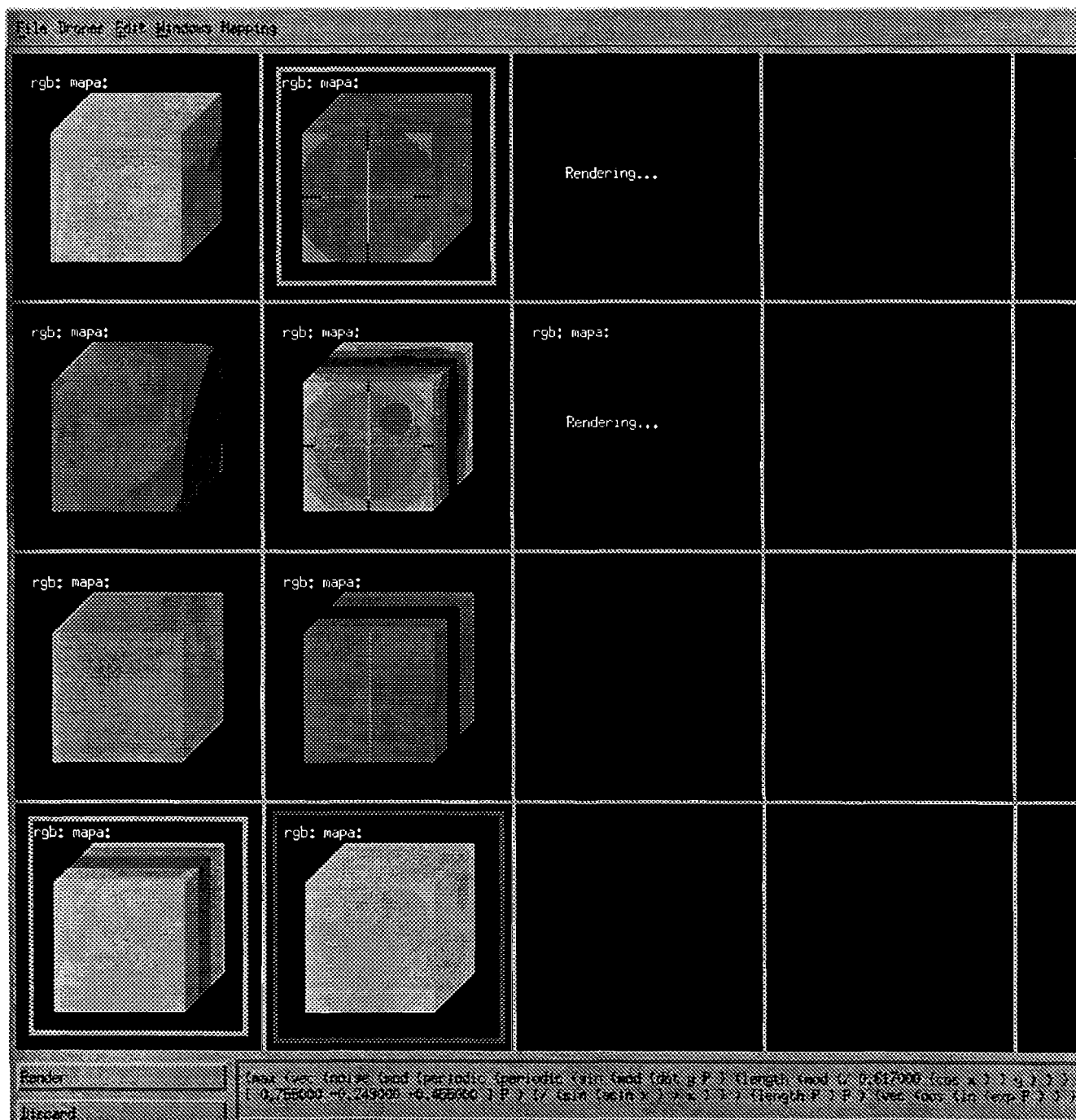


Figure 3: User's Blended Textures

RenderMan shader file.

5 Design Approach

The approach we took was to take the Sims's artificial evolution concept and extend it to generate and manipulate 3D textures in the form of s-expressions and RenderMan shaders. The requirements for our system were as follows:

The GUI had to be point-and-click for the novice user.

The GUI had to support the display of multiple images so that the user could judge their fitness easily.

The user had to be able to select image(s) to save, clear, or evolve (apply genetic operators) .

The user had to be free to initiate other operations while rendering is taking place (asynchronously).

Given that the RenderMan shader language supports the concept of 3D textures, we had to generate a "shader" from an artificially evolved arbitrary s-expression and apply it to a object for rendering.

The rendering (s-expression evaluation) had to be quick enough to keep the user interested.

The evolution of functions had to produce interesting images in a small number of iterations.

The design had to be renderer-independent or adaptable.

Given the above requirements, the TX design has the following components: GUI, Network Controller, Genetic Engine, RenderMan Interface, and Stand-alone Renderer. The TX design is master-slave with the GUI, Network Controller, Genetic Engine making up the master process and either the RenderMan Interface or Stand-alone Renderer being the slave process. See Figure 4.

5.1 GUI

The most important aspect of the GUI is the display of multiple images for fitness judgement and evolution operations. The fitness criterion in the genetic system is the user's aesthetic sense and preference. The user can clear individual windows or can pick some to keep and clear the rest. The user can select an individual texture function for editing or mutation. The user can select a pair of texture functions for crossing and blending.

When the user initiates a mutate, cross, or blend operation the s-expression(s) are sent to the Genetic Engine for processing. The Genetics Engine will return a user selectable number of results. These are then sent to the Network controller. The Network controller sends the s-expression(s) to the slave processes and returns the rendered images to the GUI for display.

MASTER PROCESS

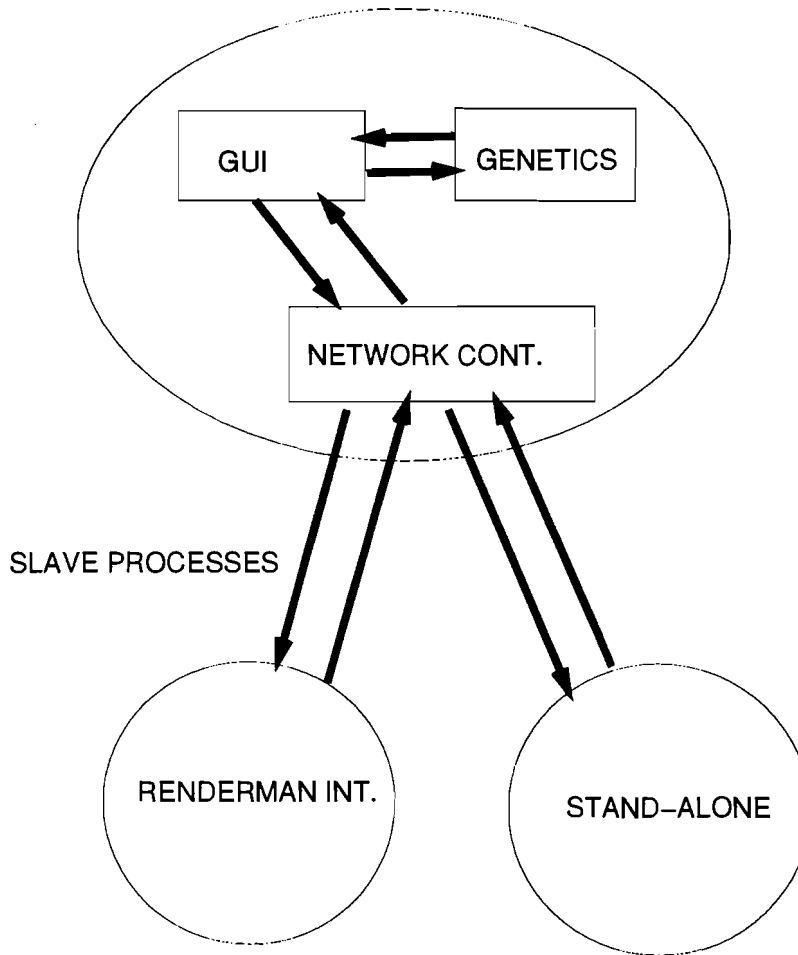


Figure 4: TX Program Architecture

The GUI also controls the following: the saving and reading of 3D textures functions and images, the reading of genetic parameters for the Genetic Engine, the reading of slave host names for the Network Controller.

5.2 Genetic Engine

Since the user is to have little input into the details of the 3D textures generated, we use a genetic algorithm to generate 3D texture functions in the form of s-expressions. S-expressions can be represented as a tree structure and as such can be manipulated and combined with other trees in many ways. There are a number of input parameters which effect the manipulation of the s-expressions, we describe in section 6.4.

See Figure 5.

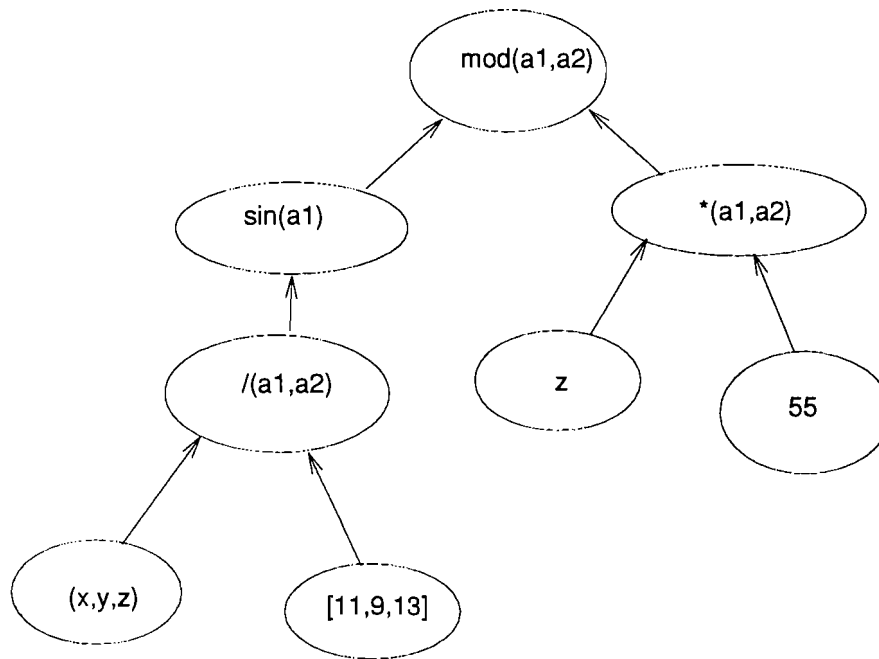


Figure 5: S-Expression As a Tree

5.3 RenderMan Interface

The rendering of a 3D texture involves converting the s-expression into some form the renderer can handle, and then binding or applying the texture to some object in some scene, and finally rendering.

The RenderMan Interface supports the concept of procedural modeling (of which 3D textures are a subset) through the use of “shaders” and their shading language. Surface shaders are used to calculate the color and/or opacity during rendering at points on the surface on an object. There are numerous global variables that can be used in calculating the color and opacity. The most important variables for our purposes are the x, y, and z values of the current surface point being rendered in object space. Knowing this point is what makes 3D textures possible.

We want to construct a shader based on an arbitrary symbolic lisp expression which will determine the surface color based on the available x, y, and z values in object space. We also want to include some of the “normal” factors used in calculating the surface color, ambient, diffuse, and specular factors.

We must translate a s-expression to the RenderMan shader language. The s-expression functions must be somewhat polymorphic, because their arguments can be either scalars, vectors, or the output of any other function. All our functions return either a scalar or a vector. The RenderMan shader language supports some polymorphic functions but most of its functions are not polymorphic. In order to work around this, we constructed functions to handle all the possible types of arguments.

5.4 Network Controller

Several requirements made a master-slave design appealing. The master-slave design allows for asynchronous operation, renderer independence, and simultaneous rendering of images. The Network Controller is used to distribute the s-expressions to remote machines to be rendered by a slave process.

The “master” process starts and controls “slave” processes (drones), which do the evaluation (expression) of a genotype into a phenotype. These drones are sent a symbolic lisp expression. They create an image based on this expression and send the image back to the master process.

The master process and the slave rendering process must be robust enough to handle each other starting and stopping. We use Quahog to control the master-slave interaction, which helps us address these difficulties [Baz93]. Quahog will kill processes it has started if it detects that the load on the remote host is over some level. It is a very “nice” program.

The number and names of the remote hosts to be used are user configurable. The user should be able to kill drones/slaves and start new ones at any time from the GUI. When master process dies, the drones/slaves must be killed. Quahog deals with these aspects of master/slave control automatically.

5.5 Renderer Independence

An alternative to using RenderMan was to develop a extremely limited stand-alone renderer, for a fixed shape that could evaluate the s-expression at each visible surface point. This would demonstrate renderer-independence, and would also potentially be faster than the RenderMan render. It would also provide independence from the RenderMan-compliant renderer in case it was not available. in some environment. It would, however, be much less flexible than the RenderMan renderer. Nonetheless, we implemented an alternate renderer to work on the standard object in the TX window (a cube).

6 Texture Space

One can view an individual 3D texture function in s-expression form as a sample (test) point in the space of all 3D textures. Theoretically a s-expression can represent any possible real-valued function. Sampling of texture space by s-expression generation is controlled by the development rules (genetic operations) used in combining a finite set of functions with variables and constants.

6.1 Genetics-Artificial Evolution

Sims in [Sim91] used a genetic algorithm based on Artificial Evolution which manipulated symbolic lisp expressions (s-expressions) to generate plant structures, 2D images, and solid textures. A genetic algorithm is a searching technique that generates test

points by “evolution” (random variation and selection). Artificial Evolution has as its components genotype, expression, phenotype, and a selection-fitness criterion.

A *genotype* is the encoded genetic information; in our work the genotype is a s-expression. *Expression* is the process by which the phenotype is generated from the genotype. For us, expression is the rendering of the s-expression applied to some object. A *phenotype* is the form that results from the development rules and the genotype and which for us is the resulting image. The *selection-fitness criterion* is the process that determines the likelihood of survival; for us this is human aesthetic opinion.

Normally a genetic algorithm is used to search for a global optimum in a very large search space. It is an iterative process, with the generation of new test points from old ones. Usually there are several “parallel test paths - multiple test points.” In the case where human opinion is used as the selection criteria there is no single global optimum to be found. So the genetic algorithm should be “geared” toward traversing the search space and finding all local optima (interesting solutions) while avoiding getting caught in any particular local area of search space. However one would hope also to be able to explore local areas sufficiently to “refine” an interesting solution.

The generation and display of 3D textures involves the following steps, s-expression generation, color mapping of the s-expression into a 3D texture, and sampling of the 3D texture. Figure 6 shows these steps.

The following describes the design decisions we made in the following areas based on our experience with TX :

the genetics operations

the s-expression language

the genetic parameters effecting how the first two interact

the mapping of an s-expression 3 space into a 3D texture (color)

the sampling of our 3D texture

how we start the search

6.2 Genetic Operations

In order to avoid getting caught in a region of search space one must be able to make substantial changes to the s-expression. We support three operations on an s-expression: mutation, crossing, and, blending. For each genetic operation the s-expression is converted into a tree structure which is then manipulated. Each of these operations use some random operation to generate a new s-expression from an existing s-expression(s). Changing the root function is the most obvious way of making large jumps in search space. This is not always the case if the new root’s range and domain are similar to the previous root’s. Generally changes higher in the tree (closer to the root) cause the more drastic change in search space sampling.

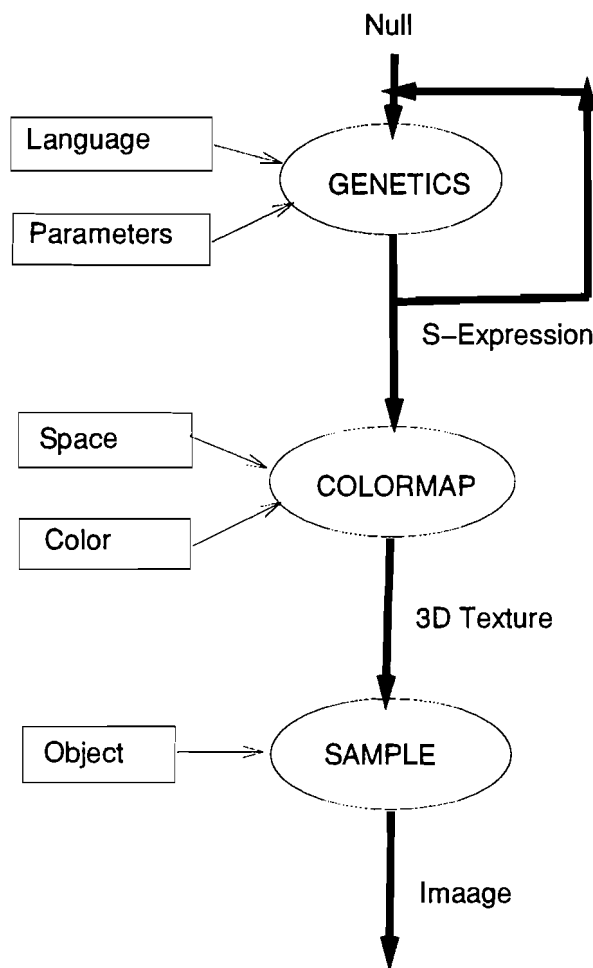


Figure 6: Texture Space Search Process

Mutation modifies a single node in a s-expression by replacing a function with another, changing a value, or replacing a variable. Replacing a function can change the nodes subtree considerably if the arity of the functions are not the same.

Crossing takes two existing s-expression and selects randomly some point in each tree and swaps sub-trees. Crossing can also change the subtrees considerably if the arity of the nodes are not the same. Arguments may have to be generated or dropped.

Blending takes two existing s-expression and selects randomly a function with 2 arguments to be the parent function of the two selected.

The genetic operations mutate, cross, and blend traverse the search space in different manners. Each is really more than a single method: there are several ways to mutate, cross, and blend. Each has some random element and some parameters to provide some control to the user. These parameters often specify the weights of the possible random choices.

An important consideration in using a particular method is whether it will make a local change or global change (new local space). Another is whether it adds to the length of the s-expression (number of nodes) - this is a performance consideration. The longer the s-expression, the slower will be its evaluation, since this “function” is evaluated many times over the surface of an object. In all operations, if a new node is needed it is chosen randomly based on some user input or default weights.

When a random node in a tree is selected, a *downward mutate probability* is used to determine approximately how far down the tree will be the node selected. A user input parameter is used to specify the downward probability type: low, middle, or high.

When a random node needs to be created, the first random choice is what node type it will be (function, variable, scalar, vector), and then a random choice is made within that node type. User input parameters are used to specify the node type weights.

Mutation

There are many ways to mutate an s-expression. We support the following: adjust constant, replace a function, rotate vector values, promote a subtree, demote a subtree, replace a subtree.

If mutation is called on a null expression then a random tree is generated. This is covered more under “Getting Started” below. A user input parameter is used to specify the method of mutation or whether it is to be randomly determined.

Each of the mutation methods randomly determines where in a tree a mutation is to take place. The random selection of a node involves traversing the tree and at each node randomly determining if this is the node to be mutated.

The *adjust constant* mutation alters a scalar value or vector value by adding a percentage of its value. *Replacing a function* randomly chooses a function and replaces it with a new function while adding or deleting arguments as necessary. *Rotating* rotates the values of a vector. *Replacing a subtree* randomly chooses a node and replaces it with a new node and generates arguments as necessary. *Demotion* involves adding/inserting a new function somewhere in a tree. It starts at the root and recursively walks the tree until the function is inserted based on the mutation probability. *Promotion* involves removing a function somewhere in a tree and promoting its children to become arguments to its parent.

Crossing

There are many ways to cross two s-expressions. We support the following: close and crude.

A user/default specified parameter is used to determine which method is used. If random selection is chosen then one of the methods is chosen randomly based on user/default weights. A user/default specified parameter is used to determine if is mutation is to occur during crossing.

Cross crude picks a random node in each tree based on a depth first numbering of nodes. The subtrees at those node are swapped to produce two new s-expressions. *Cross close* picks randomly which parent gives (donates) a function node recursively. If either of the parents are not function nodes the chosen parent is copied. This produces one result at a time.

Blending

Blending randomly chooses a new parent function of arity two and sets the two s-expressions as its arguments. This method will result in an s-expression with length equal to the lengths of the “parents” + 1.

6.3 S-Expression Language

A 3D texture function in the form of an s-expression calculates the color given any point in object space. The artificial evolution process creates texture functions by randomly combining a finite set of functions and variables with scalar and vector values.

The 3D texture “language” includes the following:

Functions: finite set of functions

Variables: x, y, z , and P - the point (x, y, z)

Scalars: real numbers

Literals: vector of 3 scalars (e.g., $[1.0, 2.0, 3.0]$)

The functions in the language need to be polymorphic since they can be combined in any way. Generally every function must be able to take a scalar or a vector of size 3 as its arguments. In some cases the type of the arguments can only be determined in a recursive manner by determining the types of the arguments arguments and so on. In some cases the output type of a function may be independent of the types of its arguments. In some cases what the functions behavior is for a given type of argument may not be obvious and needs to be defined.

Since the functions can be combined in any way and the argument values are a function of the object space sampling (shape) argument, range checks need to be added and default values assigned to some functions.

One wants a small but powerful set of functions. One thing to consider is that many basic arithmetic and trigonometric functions have particularly distinctive behavior around 0.0. Many have +/- symmetry, or result in 0 or infinite values. This may or may not be desirable. Since the 3D texture is usually defined in object space the object’s surface will probability surround the origin. Some functions were developed so that they would not have distinctive behavior around the origin or axis.

Our function set contains the following arithmetic functions : `mult(x,y)`, `add(x,y)`, `sub(x,y)`, `div(x,y)`, `mod(x,y)`.

Our function set contains the following vector functions : `length(x)`, `normalize(x)`, `distance(x,y)`, `rotate(x)`, `dot(x,y)`, `vec(x,y,z)`.

Our function set contains the following trigonometric functions : `periodic(x,y,z)`, `sin(x)`, `asin(x)`, `acos(x)`, `cos(x)`, `atan(x)`, `tan(x)`.

Our function set contains the following mathematical functions : `sqrt(x)`, `exp(x)`, `ln(x)`, `pow(x,y)`.

Our function set contains the following discrete functions : `abs(x)`, `min(x,y)`, `max(x,y)`, `round(x)`.

Our function set contains the following random functions : `noise(x)`, `invnoise(x)`.

While some of the functions chosen are supported in RenderMan Shader Language, shader functions had to be written to handle the range checks, and type conversions (i.e., some semantic interpretation). These support functions are pre-compiled and visible to the s-expression-based shader function. Note that we make use of RenderMan's noise function in our RenderMan rendering and our own noise function in the stand-alone renderer. These will not give identical results.

6.4 Genetic Parameters

Through much use of TX we have developed some insights into what kind of s-expressions lead to the evolution of interesting patterns and therefore what values for some of the input parameters work well. There are genetic parameters for the following:

Node Type Weights

Function Weights

Downward Probability

Mutate Rate

Mutate Method Choice

Mutate Method Weights

Cross Method Choice

Cross Method Weights

Mutate When Crossing Flag

As will be described in "Getting Started", sometimes there are conflicting requirements that make picking the node type weights difficult. We prefer to weight node types function and variable higher to get the user off to a good start.

We have some function weight preferences for periodic and noise functions (see “Sampling”). Since a 3D texture is defined in object space, the object's surface will probability surround the origin. Periodic and noise functions won't have a noticeable origin behavior if evaluated at values much greater than one. We added a periodic function that would not have distinctive behavior around the origin or axis and be a little more high level than the trigonometric functions.

$$\text{periodic}(a1, a2, a3) = 0.5 * \sin(a1.a2 + a3).$$

An example of an interesting texture produced from noise and periodic functions is

```
vec(periodic( [k1 k2 k3]. noise([n1 n2 n3] * [x y z]) + c),
    periodic( [k1 k2 k3]. noise([n1 n2 n3] * [x y z]) + c),
    periodic( [k1 k2 k3]. noise([n1 n2 n3] * [x y z]) + c))
```

with color mapping RGB and space mapping $\text{mod}(x, 1.0)$.

See Figure 7.

6.5 Mapping 3-Space to Color Space

At this point we have a s-expression that is a function from one 3 space to another 3 space. It is not yet a 3D texture. For this to be a 3D texture each point in this new 3 space needs to be mapped or assigned a color value.

The output of such a s-expression from the above set of functions could have any range of values. Most color spaces are only defined in the range 0.0 to 1.0, and most systems then map this to discrete RGB values. Therefore, one must map the output of the s-expression into some limited range like 0.0 to 1.0; This can be thought of as four step calculation.

3-space to 3-space: given (x, y, z) in 3-space, use the 3D texture function to calculate a new (x, y, z) -point.

3-space to 0.0-1.0 3-space: map the new (x, y, z) into limited 3-space.

limited 3-space to color space: map this limited (x, y, z) point into color space.

color space to RGB: map from a given color space to discrete RGB.

The third step for RGB color space would be red = x , green = y , blue = z . The third step for HSV color space would be hue = x , saturation = y , value = z . The last step is system dependent.

Notice that in order to avoid the object's x -dimension controlling the red through the third step in a RGB color space, the texture's x -dimension should be a function of x, y , and z . This would make red a function of the object's surface x, y , and z , not just x . More generally, we want $[r, g, b] = [f(x, y, z), g(x, y, z), h(x, y, z)]$. We were lacking

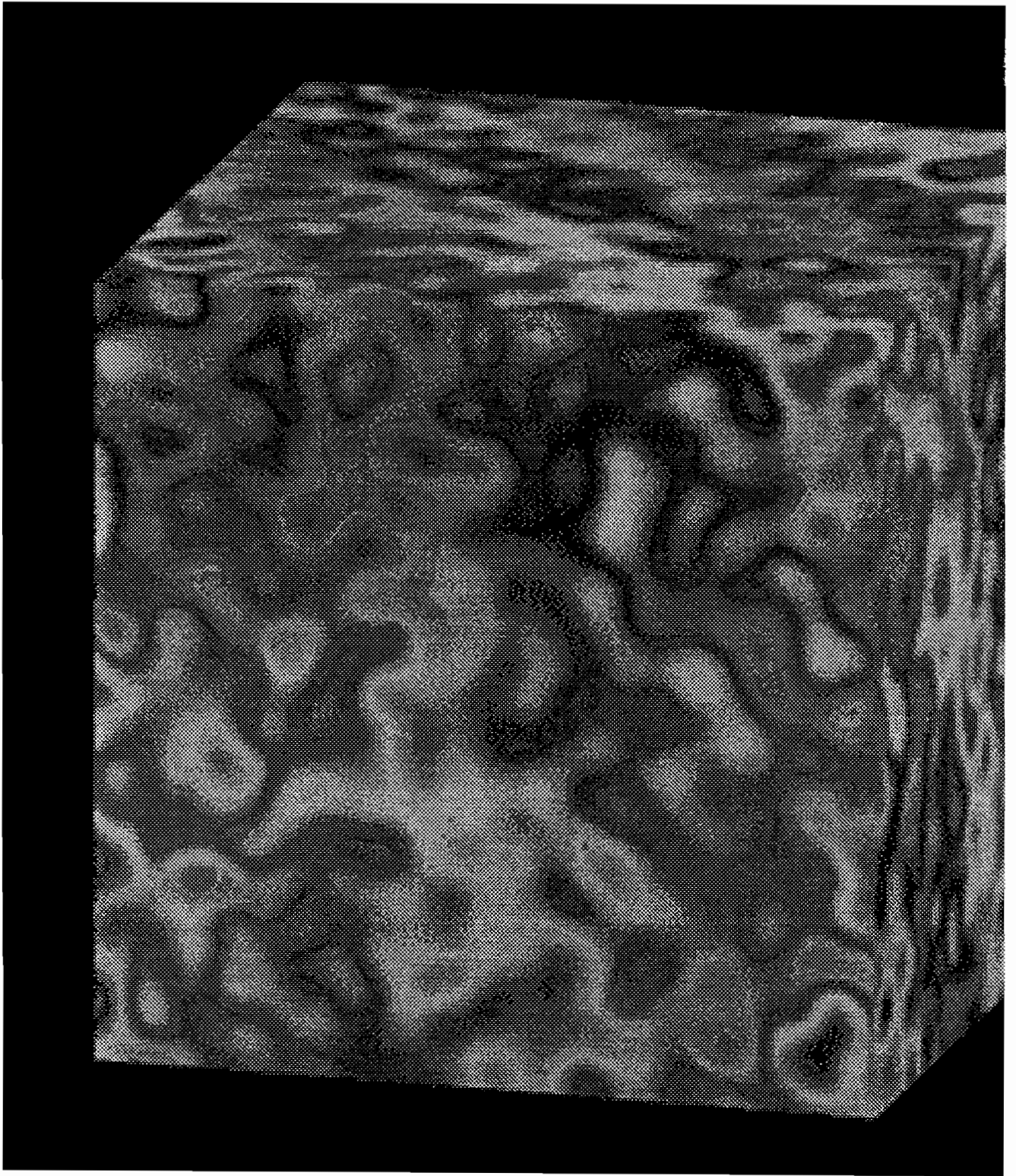


Figure 7: A Periodic Noisy Texture

a function which took three arguments and produced a vector output. This is why we added the *vec()* function.

We use this as the parent of all null mutations. A null mutation will then generate random subtrees for each argument. This does not guarantee that each component will be a function of x, y, z but it is a start (see “Getting Started” below).

We have provided the user with control over the space and color mapping. The user may want to be able to view the same 3D texture in different color spaces. A space mapping could be any function which is has infinite domain and range limited to 0.0 to 1.0. Our selection of space mappings include:

$$abs(x)/(1.0 + abs(x))$$

$$mod(x, 1.0)$$

$$abs(x)$$

$$0.5 * (sin(x) + 1.0)$$

$$noise(x)$$

See Figure 8.

Some observations about the space mappings are worth noting. The first space mapping ($abs(x)/(1.0 + abs(x))$) will compress all s-expression values from 1.0 to infinity to values 0.5 to 1.0. This is like a high pass filter. The second space mapping ($mod(x, 1.0)$) is periodic and subject to aliasing and can produce distinct banding-stripping. This often produces desirable results. The fourth space mapping ($0.5 * (sin(x) + 1.0)$) is also periodic and subject to aliasing but produces a smoother image.

We now have a *3D texture*, a function that defines a color for any point in an objects 3-space. Now how do we evaluate it and judge its fitness? Since we can't view all of it we must sample it somehow and display the sampling.

6.6 3D Texture Sampling

Sampling of any 3 space can be done in many ways. We have two main options, sample on some surface or sample through a volume.

We give the user a view of a 3D texture by looking at a surface rendering of a particular shape within that texture's space This gives the user a sample of what the texture can produce. What is a good sample surface? We chose a cube centered at the origin, with edge-length two. A cube or a sphere seemed reasonable as a more complicated object would take longer to render. A cube was chosen because it would show the change in the texture over x, y, abz more clearly.

At what scale or at what offset/location should this sample surface be placed. Normally we do not know the extent of the object to which a shader/texture is to be applied. But since one can always scale a shader, this is not particularly important.

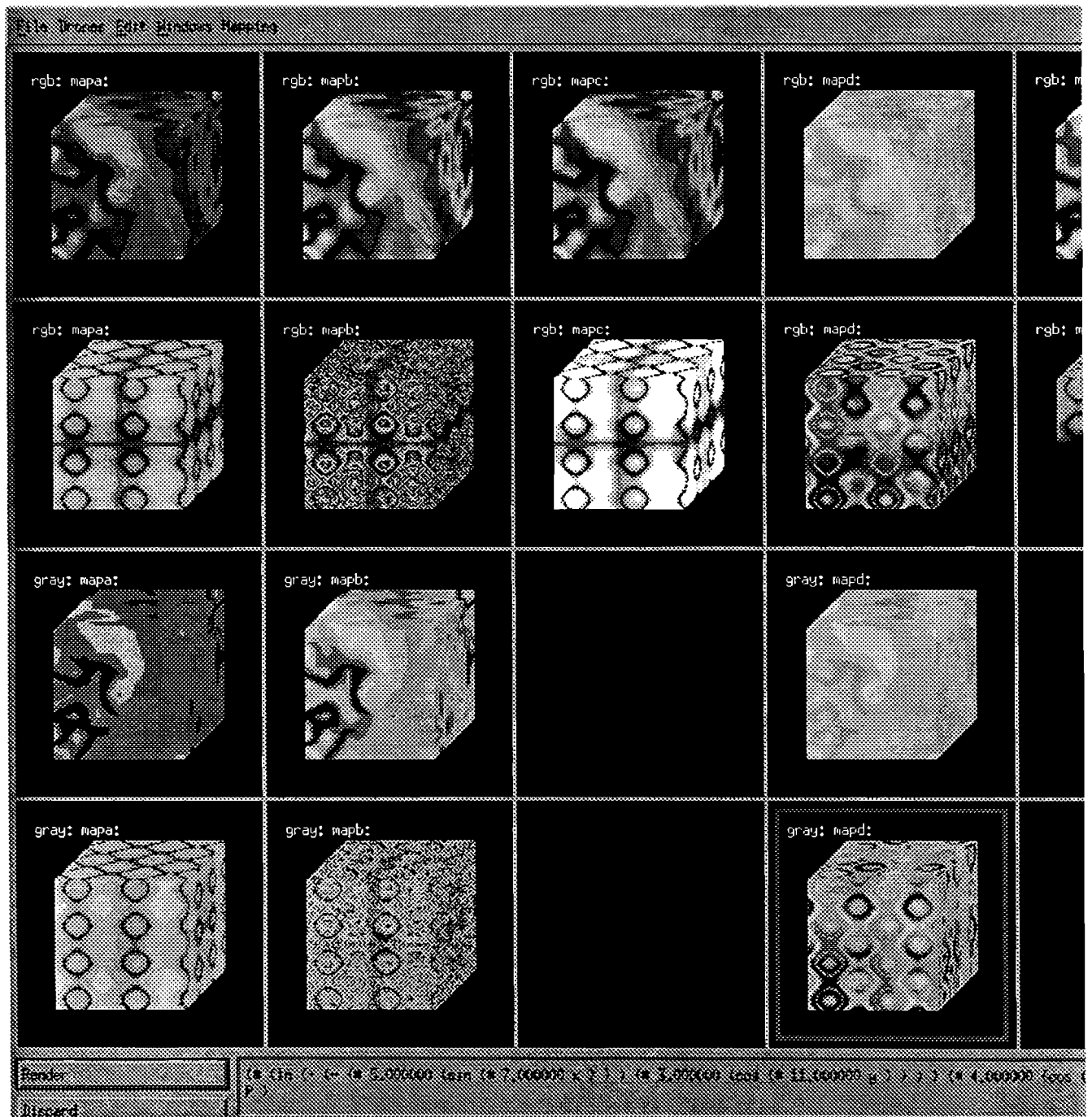


Figure 8: Color and Space Mapping

Some RenderMan shaders allow for this by having a scale instance parameter. There is a problem with surface values limited to -1 to 1. Functions like `noise()` and `sin()` will have very little change. For these functions to be worthwhile they need to have arguments that have a wide range of values. Of course too wide a range will generate aliasing.

Figure 9 show the same 3D texture a different scales.

If the computation involved in rendering the images were faster, we could give the user control of scaling and offset, so that the user could really explore a texture.

6.7 Getting Started - Something for Nothing

The genetic operations “mutate,” “cross,” and “blend” need an existing s-expression(s) to operate. So how does the user create an initial s-expression? One method is that the user can type one in, but this is not ideal for the novice. The user can also read one from a file, if some expressions have been previously created and stored. But for starting from nothing the user has only to select “Mutate” on a null image window. This will cause a random s-expression (tree) to be generated.

In generating a random tree, a random root node type can be selected based on node type weights. If the node type chosen is a function, then the arguments are chosen in the same manner and so on.

There are some problems with this method. One is that if the function node weight is too low, then the s-expressions generated will be very short and uninteresting. If the function node weight is too high, the s-expressions generated will be very long and be slow to render.

Another problem is that if the variable weight is too low, the s-expression may not be a function of x, y , or z and therefore end up being constant. If the variable node weight is too high, the s-expressions generated may not be very interesting because of the lack of constants as some functions are not very interesting over the unscaled values of x, y , or z (-1.0 to 1.0).

Aside from adjusting the weights of the node type, we have tried a few other methods for generating an initial s-expression. One was to always pick a function to be the root node, then use the random subtree generation for its arguments. All this does is guarantee that at least one function is in the s-expression.

Another method was to make the `vec(x, y, z)` function the root function, and then use the random subtree generation for its arguments. The advantage of this is that since standard color spaces are three dimensional, this guarantees that each dimension is a separate function (of x, y , or z hopefully).

Once some good s-expressions are generated, the user can then apply the mutate, cross, and blend operations on them. Cross and mutate usually don't result in s-expressions much different in length from the original s-expressions, while blend does.

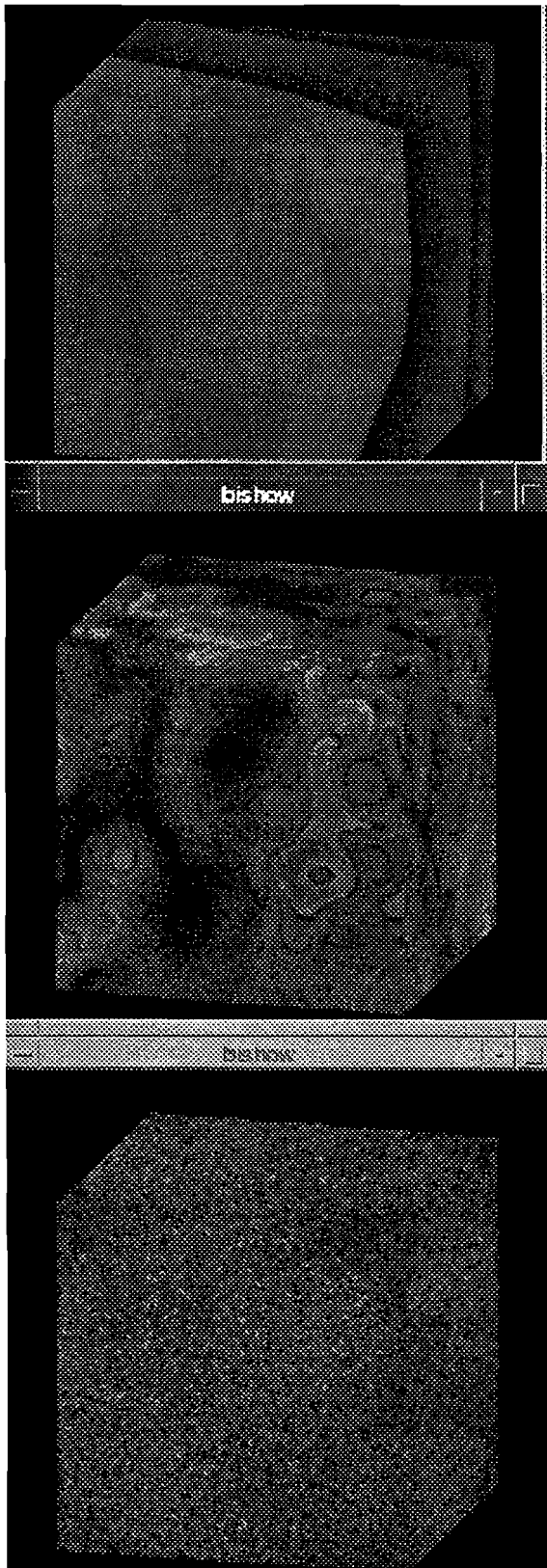


Figure 9: Scaling the Sampling of a Texture

7 Results

Our results are a combination of performance, ease of use, flexibility (re-usability) and quality of images generated (fitness).

TX's performance can be measured in two ways: (i) How long does the user have to wait for individual(new) textures to be displayed?, and (ii) how long does it take (number of iterations) before interesting results are produced?.

For the RenderMan renderer version, total user wait time includes the following : time for master to send s-expr to drone, time for slave to parse s-expr and generate shader file, time to compile shader, time to render-BI file, time for slave to read BI file, time to send image to master, time for master to display image. The RenderMan renderer version involves three machines: the master process host, the slave process host, and the RenderMan renderer host. The RenderMan renderer version takes about 30 seconds to return an image for an moderately complex s-expression with the slave process and the RenderMan renderer on a Sun4m running 4.1.3. With the slave process running on a Sun4c it takes about twice that. About 20 of the 30 seconds is for rendering; the rest is for the shader generation, shader compilation, and networking.

The stand-alone version only involves two machines, the the master process host and the slave process host. The stand-alone version takes about a third to a half the time of the of RenderMan renderer version, and certainly could be faster with some more work. The number of images that can be generated simultaneously with the stand-alone version is equal to the number of drone-slave processes.

How many iterations of selecting 3D textures and operators does it take to get interesting results starting with null s-expressions? This is subjective but does depend on the number of drones-slave processes. The screen shown in Figure 10. took about four iterations starting with null s-expressions and with five children (s-expressions) created per genetic operation.

7.1 Examples

How interesting are the 3D textures that TX can generate? That is, of course, up to the individual, but we will give a few examples of interesting textures.

Figure 11. is from the s-expression

```
(mod (pow [ 1.50 2.90 1.90 ] (length P ) )  
      (vec (mod (* [ 2.00 4.0 3.0 ] (noise (* 3.00 P ) ) ) P )  
            (periodic (* [ 7.0 3.0 5.0 ] (noise (* 3.0 P ) ) ) P 3.0 )  
            (- (* [ 11.00 0.00 5.00 ] (noise (* 3.0 P ) ) ) P ) ) )
```

with space mapping $\text{mod}(x, 1.0)$ and with color mapping RGB.

Figure 12. is from the s-expression

```
(* (ln (+ (- (* 5.00 (sin (* 7.00 x ) ) )
```

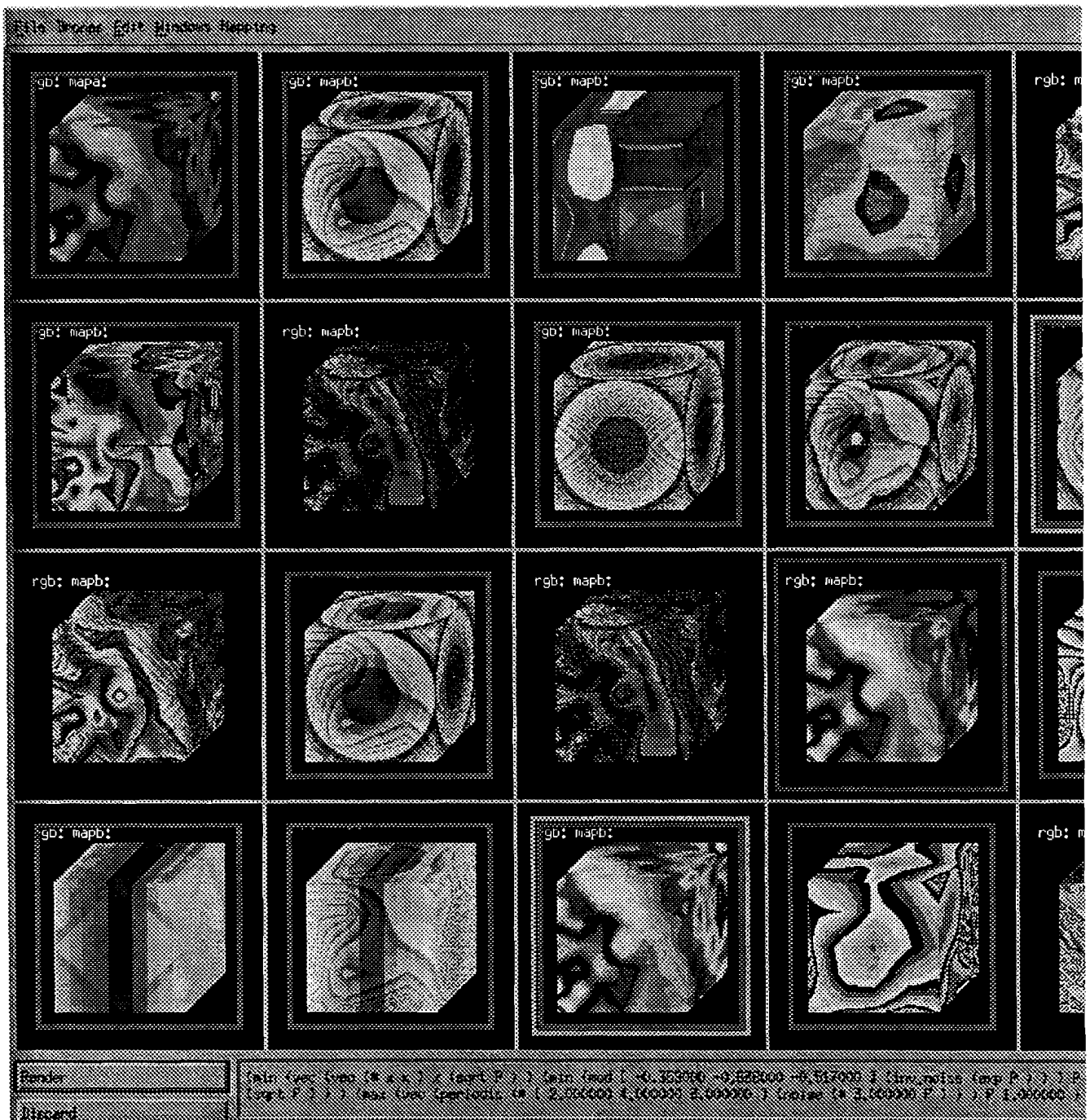


Figure 10: Example Session

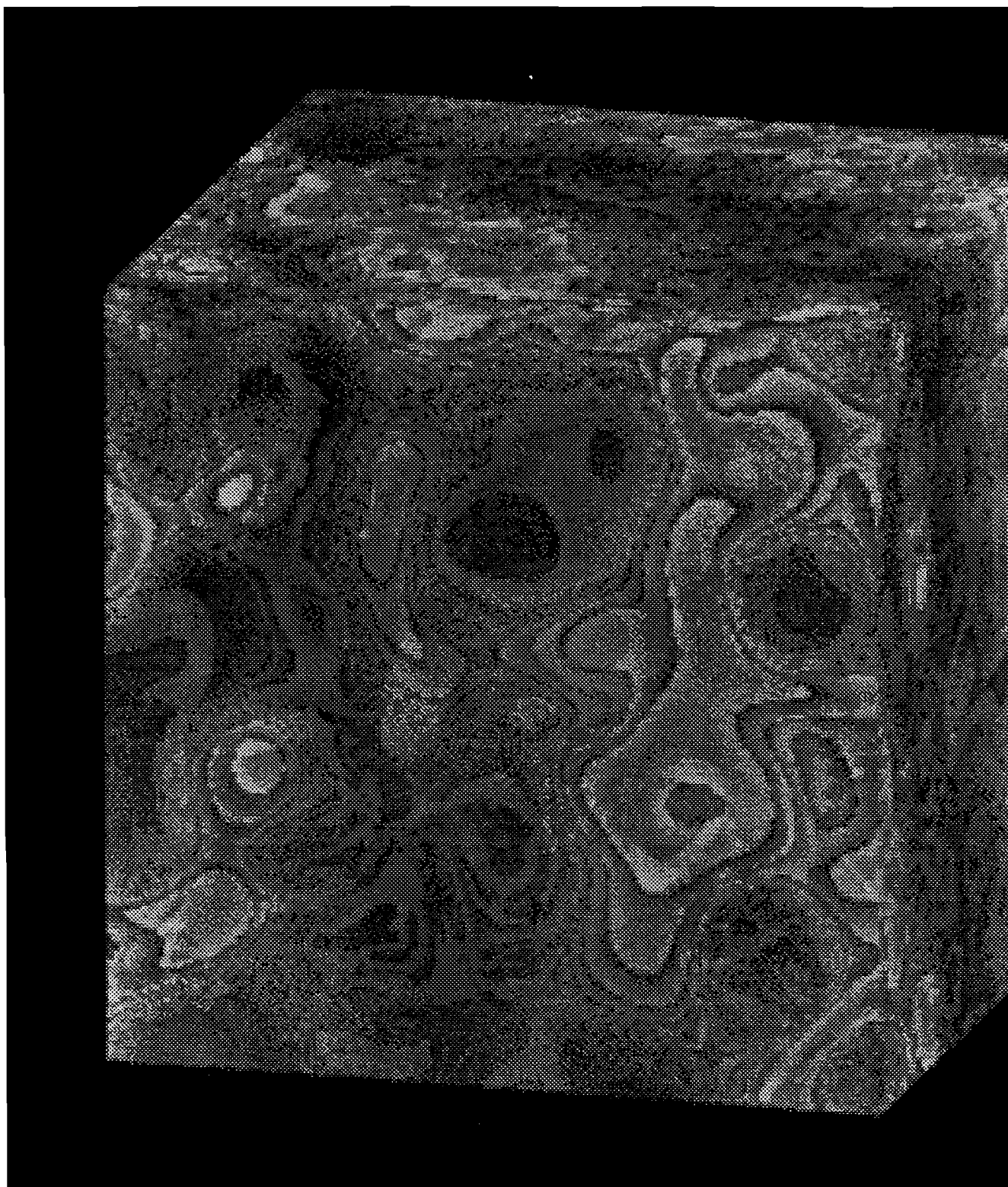


Figure 11: Example Texture

```

      (* 3.00 (cos (* 11.00 y ) ) ) )
    (* 4.00 (cos (* 9.00 z ) ) ) )
  P )

```

with space mapping $\text{mod}(x, 1.0)$ and with color mapping RGB.

is from the s-expression Figure 13.

```

(* 10.000000 (+ (* [ 3.00 3.00 3.00 ] (normalize P ) )
  (/ (acos (/ x (sqrt (+ (* x x ) (* z z ) ) ) ) )
    (* 4.00 (cos (* 5.00 z ) ) ) ) ) )

```

with space mapping $0.5 * (\sin(x) + 1.0)$ and with color mapping HSV.

One important goal was the re-usability of the textures generated. Figures 14 and 15. give some examples of the 3D textures generated applied to objects as RenderMan shaders in a more interesting scene.

7.2 Future Work

If the 3D textures could be rendered at near interactive speeds, the user could explore interesting textures with slicing, scaling, and offsetting of the sample surface. The user currently gets a limited view of the 3D texture. We could add the option to allow the user to specify what the sample surface (object) is used. We could even allow the user to input his own object in some form (polygonal or surface patch). The disadvantage of allowing the user to do this is that the more complicated the object, the longer it will take to render.

It is hard to evaluate the long term sampling of texture search space by our genetic operations and function set. We could try to develop higher level functions or add parameters to influence particular combinations of functions and arguments. We could also base our s-expression generation on some grammar. The grammar rules could more easily be made to generate specific combinations of functions and arguments.

7.3 Conclusions

TX does meet the goal of a novice user being able to generate 3D textures with little input. The user only has to select genetic operations and make aesthetic judgements on the images produced. The user also has control of the color and space mapping. The user does not have to understand the mapping functions to use them.

The time to evaluate a single s-expression and return the image is not quite fast enough to keep the user's complete interest if this is all the user is doing. It is fast enough if the user has something else to do while images are being evaluated. Because of the distributed rendering, the user could initiate the as many as 20 evaluations and the total wait time would be about the same as for one.

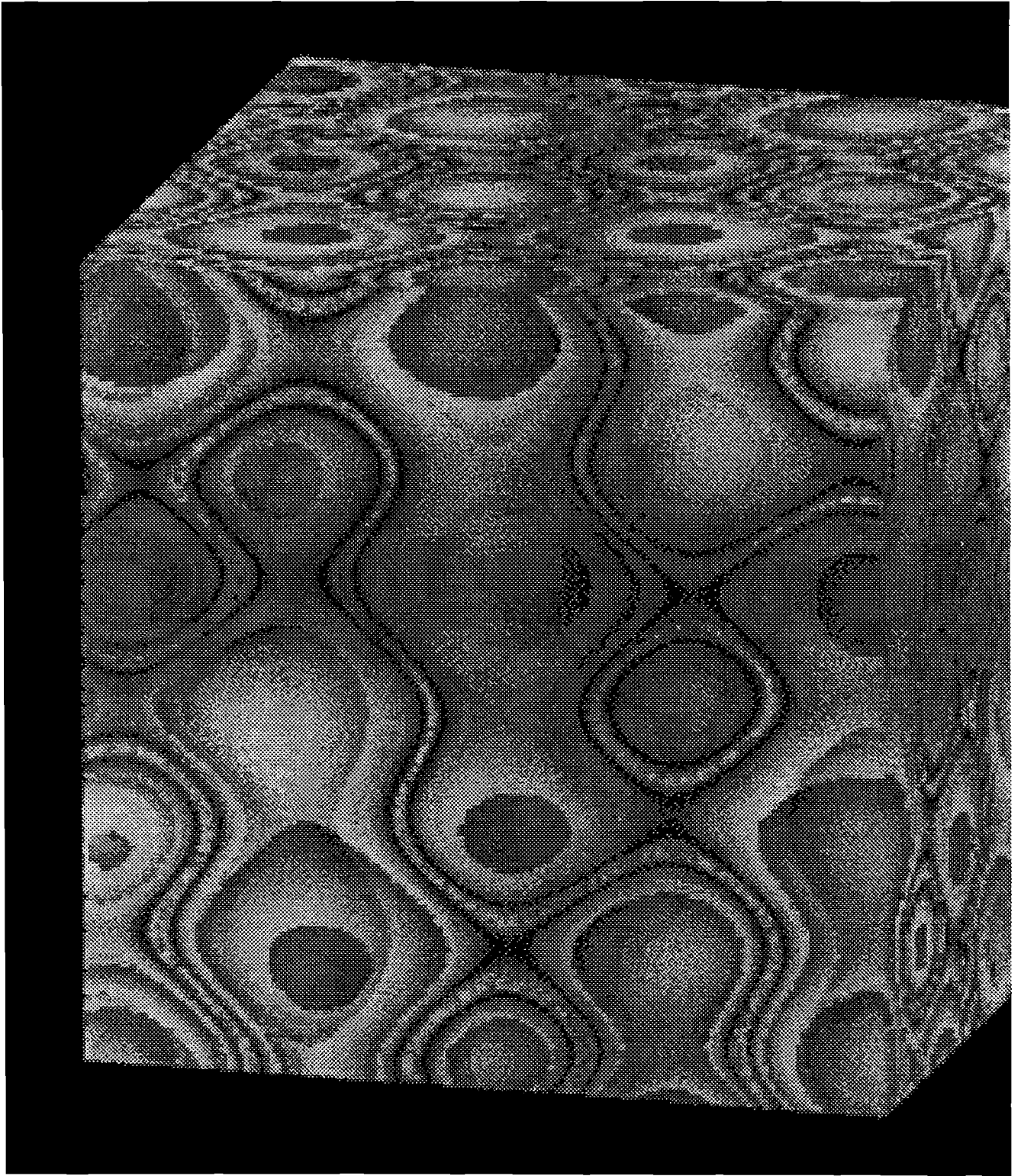


Figure 12: Example Texture

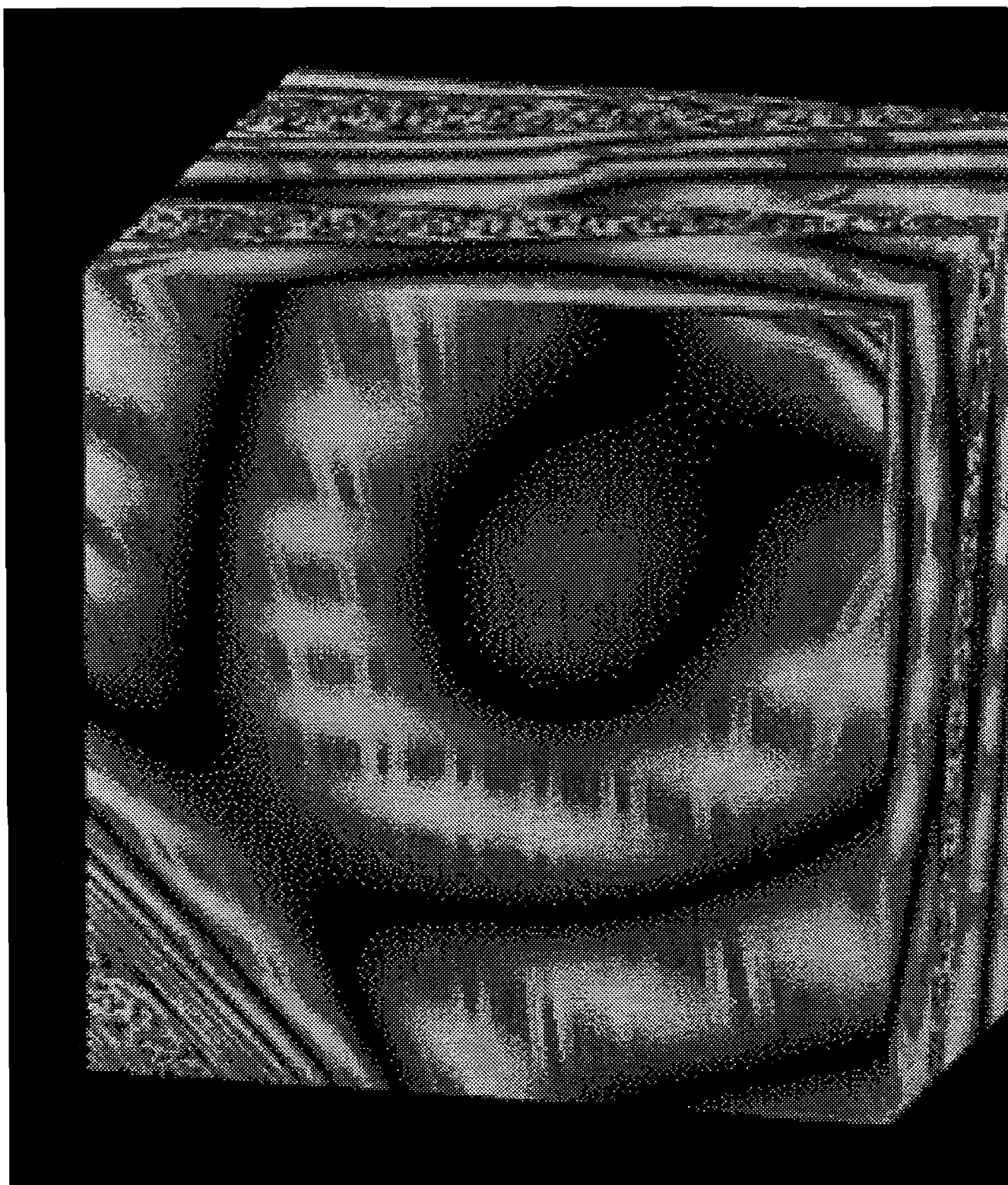


Figure 13: Example Texture



Figure 14: Example Textured Scene

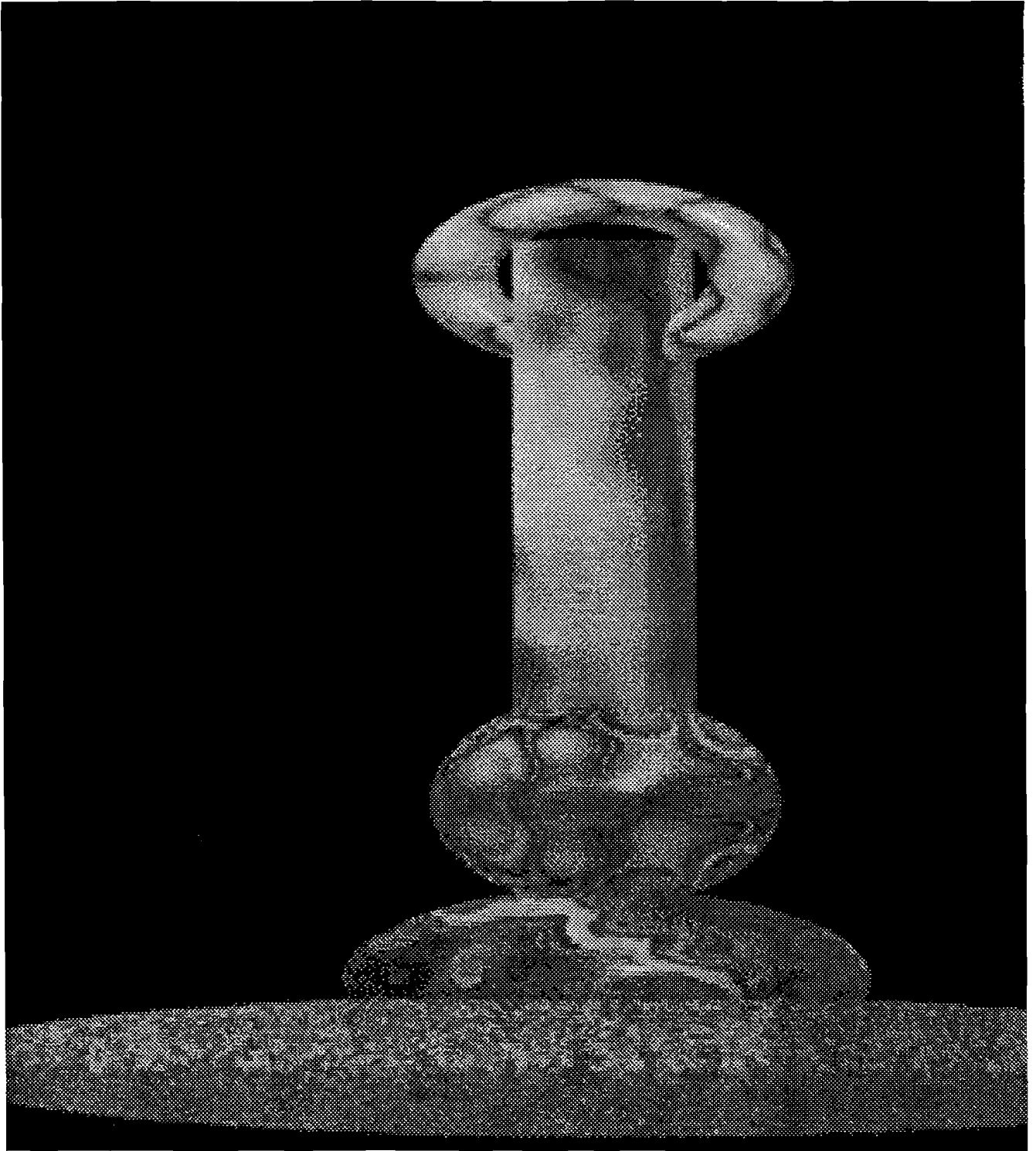


Figure 15: Example Textured Scene

It is difficult to determine if the current genetic operations and function set produce s-expressions that sample the 3D texture search space sufficiently to produce an adequate variety of 3D textures. This will be determined with long term use and cataloging of the 3D textures.

The goal of the 3D textures being re-usable is met by having the 3D textures transformed into RenderMan shaders. These can be applied to any object in a scene with a RenderMan compliant render. The s-expression format is also a generic form which can be translated into forms that other renderers might require.

7.4 Acknowledgements

I would like to thank my advisor, John Hughes, for his initial suggestion to undertake this research, his numerous suggestions along the way, and his review of my writing. I appreciate the support of the Brown Computer Graphics Group. I would also like to thank my family for their support and dedicate this in the name of my late father, John G. Krupka and my late grandfather, Dr. H. H. Schrenk.

References

- [Baz93] J. Bazik. *Quahog: Polite Remote Processing (DRAFT)*. Brown University, Providence, RI, 1993.
- [Pea85] D. Peachy. Solid Texturing of Complex Surfaces. *Computer Graphics*, 19(3):279–286, July 1985. Proceedings of SIGGRAPH '85, *published as Computer Graphics, Vol. 19, No. 3*.
- [Per85] K. Perlin. An Image Synthesiser. *Computer Graphics*, 19(3):287–296, July 1985. Proceedings of SIGGRAPH '85, *published as Computer Graphics, Vol. 19, No. 3*.
- [Per89] K. Perlin. Hypertexture. *Computer Graphics*, 23(3):253–262, July 1989. Proceedings of SIGGRAPH '89, *published as Computer Graphics, Vol. 23, No. 3*.
- [Sim91] K. Sims. Artificial Evolution for Computer Graphics. *Computer Graphics*, 25(4):319–328, July 1991. Proceedings of SIGGRAPH '90, *published as Computer Graphics, Vol. 25, No. 4*.
- [Ups85] S. Upstill. *The RenderMan Companion*. Addison-Wesley, Reading, MA, 1985.