Hypertexture

Ken Perlin
Courant Institute of the Mathematical Sciences
New York University

Eric M. Hoffert †
AT&T Pixel Machines

ABSTRACT

We model phenomena intermediate between shape and texture by using space-filling applicative functions to modulate density. The model is essentially an extension of procedural solid texture synthesis, but evaluated throughout a volumetric region instead of only at surfaces.

We have been able to obtain visually realistic representations of such shape+texture (*hypertexture*) phenomena as hair, fur, fire, glass, fluid flow and erosion effects. We show how this is done, first by describing a set of base level functions to provide basic texture and control capability, then by combining these to synthesize various phenomena.

Hypertexture exists within an intermediate region between object and not-object. We introduce a notion of *generalized boolean shape operators* to combine shapes having such a region.

Rendering is accomplished by *ray marching* from the eye point through the volume to accumulate opacity along each ray. We have implemented our hypertexture rendering algorithms on a traditional serial computer, a distributed network of computers and a coarse-grain MIMD computer. Extensions to the rendering technique incorporating refraction and reflection effects are discussed.

CR Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiprocessors - *parallel processors*; I.3.3 [**Computer Graphics**]: Picture/Image Generation - *display algorithms - viewing algorithms*; I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling - *curve, surface, solid and object representations*; I.3.7 [**Computer Graphics**]: Three Dimensional Graphics and Realism - *animation - visible line/surface algorithms*;

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

©1989 ACM-0-89791-312-4/89/007/0253 $00.75

General Terms: volume modeling, noise, turbulence, translucency, opacity, volume rendering, parallel rendering, distributed rendering

Additional Key Words and Phrases: hypertexture, generalized boolean, density modulation function (DMF), ray marching, furrier synthesis

1. Introduction

In computer graphics objects are traditionally modeled as sets having infinitesimally thin boundary surfaces. Often a computed or digitized texture or displacement is then mapped onto the surface for enhanced realism. However, there are limitations in treating object boundaries merely as surfaces.

Many objects, such as fur or woven materials, have a complex definition which is at best awkward, and at worst impossible, to describe by a surface model. For other objects, such as eroded materials or fluids, a highly complex boundary is actually an artifact of a process that is often more readily described volumetrically. Still other objects, such as flame, clouds, or smoke, don't actually have a well defined boundary surface at all.

We have found that the appearance of many such objects can be described directly by some applicative function, evaluated over a sampling of some region of R^3. Within this framework we are intuitively working with a solid block of material; visual characteristics of objects can be finely tuned by inserting numerical controls into their defining functional descriptions [3]. In that sense this work extends the procedural texture generation work of [5]. As in [5], we make use of a single controllable stochastic *noise* function together with a toolkit of shaping functions and programming constructs.

We render hypertexture by combining ideas from volume rendering ([1],[7],[8],[10],[16]) with some new extensions particularly suited to this model.

---

† Current address: Apple Computer, Cupertino, CA

1.1 Overview

In this paper, we first discuss the modeling issues of hypertexture. Instead of modeling objects as connected surfaces, we model objects as distributions of density. We describe a mechanism to generate simple base-shape density distributions. These base-shapes have a hard region where they are completely solid, and a soft region where they are indeterminate. Whenever we are in the soft region, we can apply a toolkit of shaping functions, allowing the flexibility to create and manipulate volumetric form. An analogy is to think of this region as malleable, where the user can push in, pull out, twist or otherwise deform simulated matter in a controllable manner. We also develop a CSG style scheme to combine shapes using the operators union, intersection, difference and complement.

When the foundation of modeling has been described we show how hypertexture is rendered. The renderer needs to evaluate many density samples throughout the volume since the model can become highly detailed and may contain a high degree of depth complexity. The rendering stage of the process allows a user to control the color and opacity of the object at every point in R^3 via the use of color maps. Since hypertexture rendering is relatively expensive ($O(n^3)$ with respect to image resolution), we have implemented both distributed and parallel renderers.

2. Modeling Hypertexture

In order to describe hypertexture, we need to introduce the concepts of:

- an *Object Density Function* $D(\mathbf{x})$ with range [0,1] which describes the density of a 3D shape for all points $\mathbf{x}$ throughout R^3. The *soft region* of an object consists of all $\mathbf{x}$ such that $0 < D(\mathbf{x}) < 1$.
- a *Density Modulation Function* (DMF) f_i, which is used to modulate an object's density within its soft region. Each DMF is used to control some aspect of an object's spatial characteristics; a collection of DMFs comprises a volume modeling toolkit.

Hypertexture is created by successive application of DMFs f_i to an object's $D(\mathbf{x})$:

$$H(D(\mathbf{x}), \mathbf{x}) = f_n(\ldots f_2(f_1(f_0(D(\mathbf{x})))))$$

The DMF f_i can be of three types: position-dependent, position-independent and geometry-dependent. Position-dependent DMFs are $f(\mathbf{x})$, position-independent are $f(k)$ where k is a scalar and geometry-dependent DMFs may depend on variables other than x such as density gradient in the vicinity of x.

2.1 Soft Objects

Formally, a soft object is a density function $D(\mathbf{x})$ over R^3, where D is 1.0 inside the object, 0.0 outside the object, and $0.0 < D < 1.0$ in a region of nonzero thickness in between. As an example, consider the sphere centered at c of radius r and softness s. This can be defined by the density function:

```
$D_{[\mathbf{c},r,s]}(\mathbf{x})$ :
    $r_1^2 := (r-s/2)^2$
    $r_0^2 := (r+s/2)^2$
    $r_x^2 := (x_x-c_x)^2 + (x_y-c_y)^2 + (x_z-c_z)^2$
    $D :=$ if $r_x^2 \le r_1^2$ then 1.0 else
        if $r_x^2 \ge r_0^2$ then 0.0 else $(r_0^2-r_x^2) / (r_0^2-r_1^2)$
```

where r_0 is the the outer (D=0) boundary, r_1 is the inner (D=1) boundary and r_x is the radius of the sphere at the point x.

2.2 Generalized Booleans

We extend the boolean operations of set union, set complement, set intersection and set difference[†] to soft objects A and B through their density functions $a(\mathbf{x})$ and $b(\mathbf{x})$:

- **intersection:** $A \cap B \equiv a(\mathbf{x})b(\mathbf{x})$
- **complement:** $\overline{A} \equiv 1.0 - a(\mathbf{x})$
- **difference:** $A - B = A \cap \overline{B} \equiv a(\mathbf{x}) - a(\mathbf{x})b(\mathbf{x})$
- **union:** $A \cup B = \overline{\overline{A} \cap \overline{B}} \equiv a(\mathbf{x}) + b(\mathbf{x}) - a(\mathbf{x})b(\mathbf{x})$.

As with traditional boolean shape operators, we can construct expressions of arbitrary complexity to represent combinations of different primitive shapes. By using boolean algebra, soft objects can be added or subtracted with smooth and controllable fillets at the regions where they join. Note that the size of the region where objects join can be controlled by modifying the width of each object's soft region. It should be noted that these operations don't actually form a complete boolean algebra (since from the above definitions $A \cap A \neq A$ and $A \cap \overline{A} \neq \Phi$) but do allow for standard set operations.

2.3 Base Density Modulation Functions

Here are the base level DMFs that higher-order DMFs are built upon.

[†] There is a rough analogy here to the principles of fuzzy set theory [18]. We choose algebraic sum and algebraic product for union and intersection, respectively [19], instead of the fuzzy set theoretic *min* and *max* operators [18], because the former preserve continuity. This is needed to maintain smoothness of fillets at the regions where objects join. It would be interesting also to try the *min* and *max* operators.

bias

We use bias to either push up or pull down an object's density. A function that controls mean, $bias_b$ is a power curve defined over the unit interval such that $bias_b(0)=0$, $bias_b(1/2)=b$, and $bias_b(1)=1$. By increasing or decreasing b, we can thus bias the values in an object's soft region up or down. $Bias_{0.5}$ is the identity function over [0,1].

Bias can be defined by the power function:

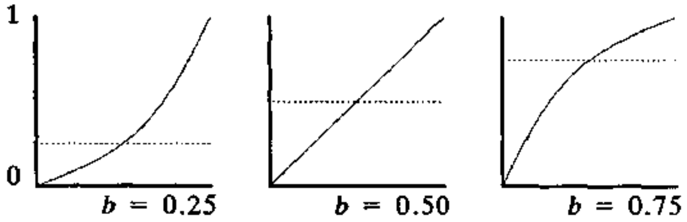
$$t^{\frac{\ln(b)}{\ln(0.5)}}$$

gain

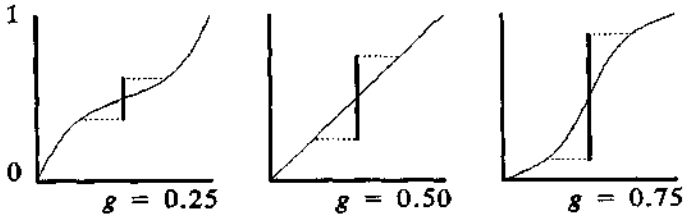
We use gain to make an object's density gradient either flatter or steeper. A function that controls variance, $gain_g$ is defined over the unit interval such that: $gain_g(0)=0$, $gain_g(1/4)=(1-g)/2$, $gain_g(1/2)=1/2$, $gain_g(3/4)=(1+g)/2$, and $gain_g(1)=1$. By increasing or decreasing g, we can thus increase or decrease the rate at which the midrange of a object's soft region goes from 0.0 to 1.0. $Gain_{0.5}$ is the identity function over [0,1].

Gain can be defined as a spline of two bias curves:

$$\begin{aligned} &\textit{if } t < 0.5 \textit{ then } bias_{1-g}(2t)\ /\ 2 \\ &\qquad\qquad \textit{else } 1 - bias_{1-g}(2-2t)\ /\ 2 \end{aligned}$$



Different *bias* curves



Different *gain* curves

noise

An approximation to white noise band-limited to a single octave, *noise* [5] allows us to introduce randomness into the digital signal without sacrificing either continuity or control over spatial frequency.

We implement *noise* as a summation of pseudorandom spline knots, one for each point on the integer lattice of R^3. The knot $\Omega_{i,j,k}$ at lattice point (i,j,k) consists of a pseudorandom linear gradient $\Gamma_{i,j,k}$ weighted in each dimension by a smooth drop off function $\omega(t)^{\dagger}$:

$$\Omega_{i,j,k}(u,v,w) = \omega(u)\omega(v)\omega(w)\ (\Gamma_{i,j,k} \bullet (u,v,w))$$

where "•" denotes vector inner product. We choose for $\omega(t)$ the cubic weighting function:

$$\textit{if } |t|<1 \textit{ then } 2|t|^3-3|t|^2+1 \textit{ else } 0$$

giving the spline a support of 2 in each dimension, so that in practice for any given point in R^3 we only need to take the sum of the 2^3 nearest spline knots. Thus our *noise* implementation is defined at point (x,y,z) by:

$$\sum_{i=\lfloor x\rfloor}^{\lfloor x\rfloor+1}\ \sum_{j=\lfloor y\rfloor}^{\lfloor y\rfloor+1}\ \sum_{k=\lfloor z\rfloor}^{\lfloor z\rfloor+1} \Omega_{i,j,k}(x-i,y-j,z-k)$$

where "$\lfloor\ \rfloor$" denotes the floor function.

For speed, we implement the pseudo-random gradient by hashing (i,j,k) to create an index into a precomputed gradient table **G**:

$$\Gamma_{i,j,k} = \mathbf{G}[\phi(i+\phi(j+\phi(k)))]$$

where:

- $\phi(i) = P[i_{mod\ n}]$, where P is a precomputed array containing a pseudorandom permutation of the first n integers,
- **G** is a precomputed array of n pseudorandom vectors uniformly distributed on the unit sphere,
- n is the length of the P and **G** arrays (in practice, we find n=256 to be a reasonable value).

To ensure that each element v of **G** is uniformly distributed on the unit sphere, we employ a three step Monte Carlo

---

[†] It would have been somewhat faster and simpler to use a constant Γ (which is essentially a wavelet model), but we have found that this produces visible artifacts at the lattice points, where gradient becomes zero.

technique:

(1) generate each coordinate of v, choosing uniformly from the interval [−1,+1]
(2) if $|\mathbf{v}|>1.0$ then goto (1)
(3) normalize the length of v

It is important to use the three-fold table lookup of "$\phi(i+\phi(j+\phi(k)))$" above for hashing the three integer lattice coordinates (i,j,k), so that neighboring lattice points will not have correlated indices into the gradient table **G** (which would otherwise be visible as unsightly patterns).

turbulence

Because of its great general utility in building higher level DMFs, we also include the *turbulence* function of [5] as a base function, defined by:

$$\sum_i abs\left(\frac{1}{2^i}\, noise\,(2^i\,\mathbf{x})\right)$$

Note that this is not a true turbulence model, but merely a method of simulating the appearance of turbulent activity.

arithmetic base functions

The set of base functions is rounded out by basic mathematical routines such as the *abs* and *sine* functions, together with arithmetic and control flow operations.

2.4 Higher Level Functions (Hypertextures)

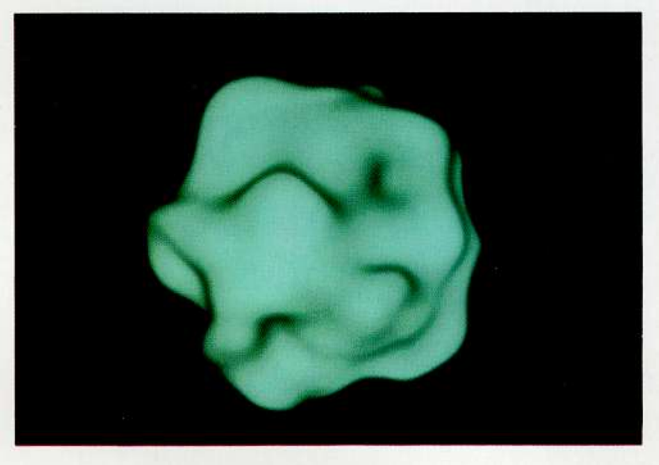
In this section, we describe how to create and combine DMFs to generate hypertextural phenomena. In most of our examples, the shape defined by the object density function of the hypertexture is very simple, such as a cube, sphere or torus. The DMFs shape the soft region of these objects; color and alpha maps determine the mapping of density to color and transparency.

Basic Noise

Figure **noisy sphere** shows a sphere with noise of frequency f and amplitude **1/f** used to scale the radius:

$$D(\mathbf{x}) = sphere(\mathbf{x}\,(1 + \frac{1}{f} noise(f\mathbf{x})))$$

The frequency controls the number of bumps on the surface; the amplitude controls their height.



noisy sphere

Varying Frequency

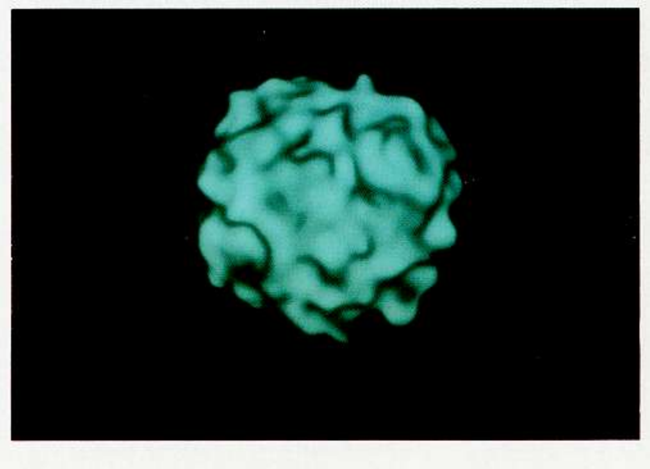
Figure **high-frequency noisy sphere** shows the same sphere with noise of twice the frequency and half the amplitude. This creates smaller but similarly shaped perturbations of the surface.

Varying Amplitude

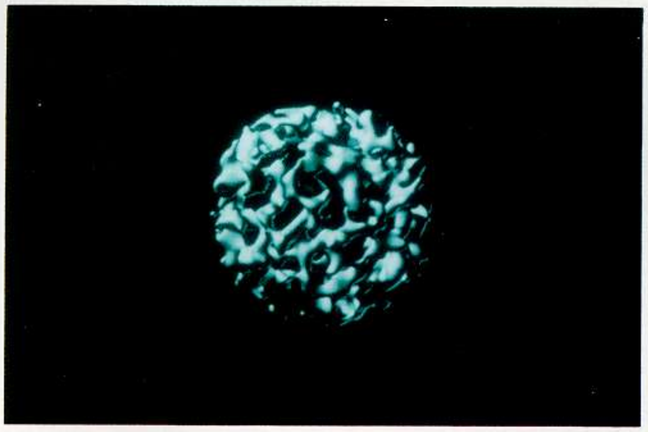
Figure **high-amplitude noisy sphere** is the same as above but with amplitude increased, so that the noise modulation dominates the shape.

Combining Frequencies

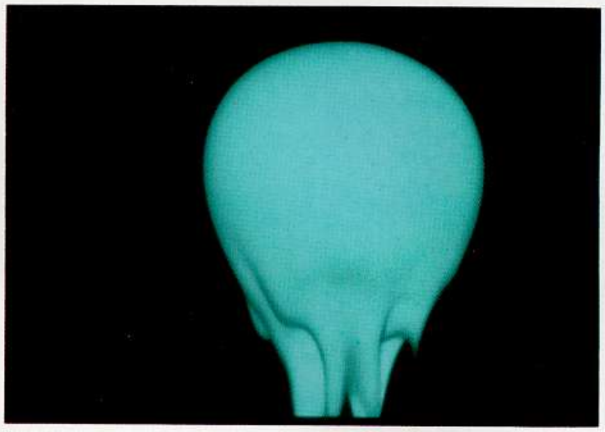
Figure **fractal sphere** shows the same sphere again, with noise of many different frequencies summed together:



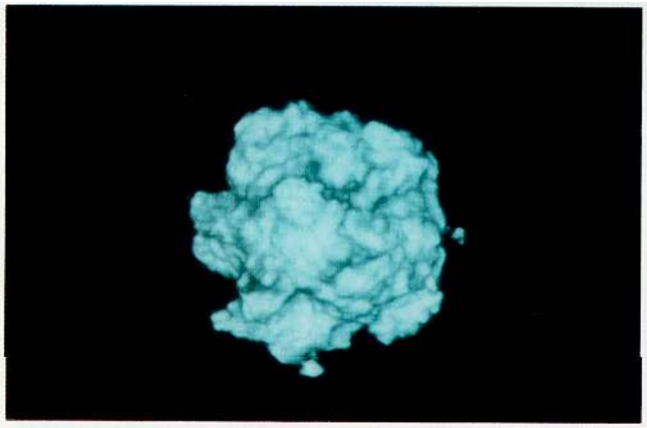
high-frequency noisy sphere



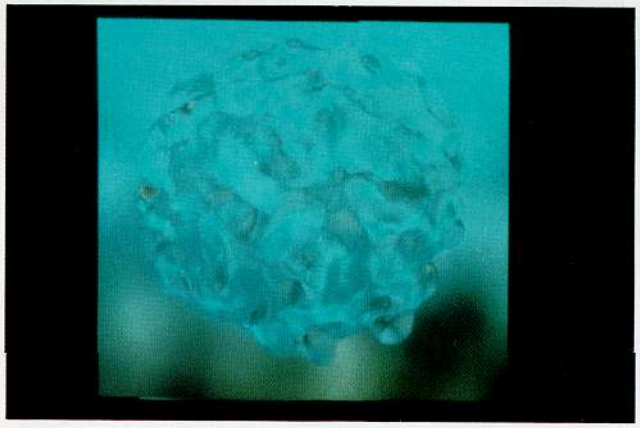
high-amplitude noisy sphere



dripping sphere



fractal sphere



blue glass

$$D(\mathbf{x}) = sphere(\mathbf{x}\ (1 + \sum_i \frac{1}{2^i f} noise(2^i f \mathbf{x})))$$

The base frequency here is f; at each step of the summation, amplitude is inversely proportional to frequency. Now the shape takes on a characteristic $\frac{1}{f}$ fractal appearance.

Shaped noise

Figure **dripping sphere** illustrates what happens when we use noise to modulate only the y component of $\mathbf{x}$ in order to simulate the appearance of dripping material.

Transparency

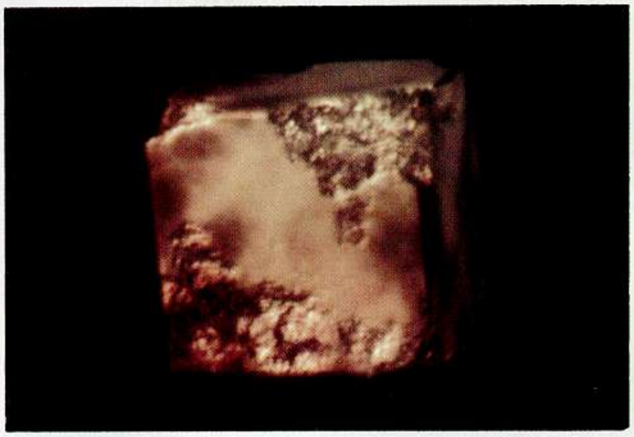
Figure **blue glass** illustrates the notion of refractive hypertexture. In this case, rays are bent following Snell's law [20] whenever the refractive index encountered by the renderer at a sample is different from its value at the previous sample. We add an extra channel to the renderer's color table to modulate refractive index as a function of density. In this example we apply it to one of the noisy spheres seen earlier.

Erosion

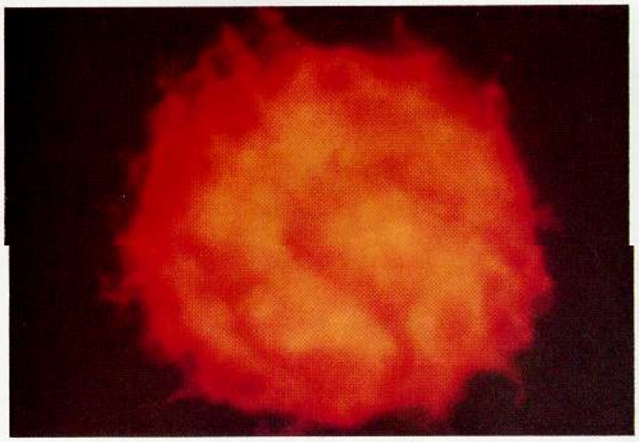
Figure **eroded cube** is an example of an erosion model. We use generalized booleans to generate this composite shape by applying the **intersection** operator to combine a fractal sphere with a cube. The turbulence function is also used here, to create color variations through the hypertexture as in [5].

Fire

Figure **fire ball** was created by the density function:



eroded cube



fire ball

$$D(\mathbf{x}) = sphere(\mathbf{x}\,(1 + turbulence(\mathbf{x})))$$

The color map is structured in this case so that low densities map to red, higher densities to yellow. This is a direct extension to three dimensions of the flame model used to create the **Solar Corona** of [5].

We now describe an example of hypertexture in greater detail, to give a sense of how these algorithms are developed.

Hair and Fur

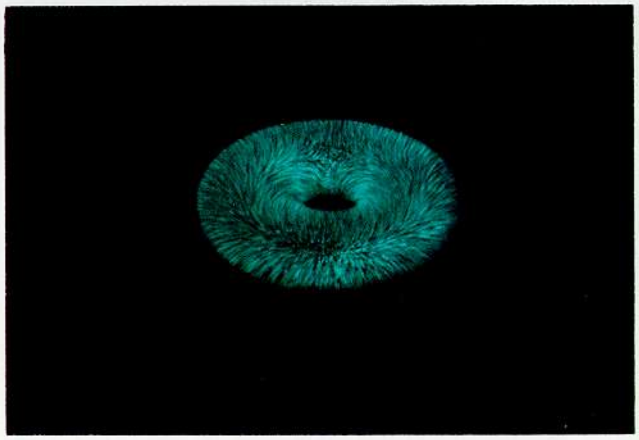
We create furlike hypertexture in stages. First we start with a soft object defined by $D(\mathbf{x})$. The fur will exist in the object's soft region, each filament growing out from the inner boundary (where $D = 1.0$) towards the outer boundary (where $D = 0.0$).

We first project each point **x** in the soft region perpendicularly down to the inner solid boundary surface using the geometry-dependent function **project**. At the surface we compute *noise* of high frequency *freq*, and use it to modulate the object density $D(\mathbf{x})$, as follows:

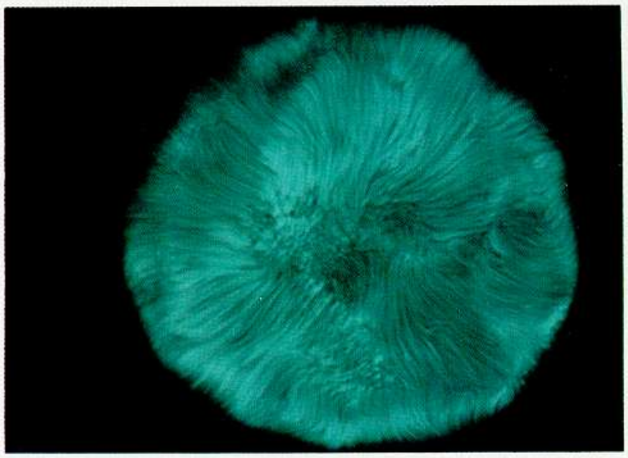
$$s := noise(freq * \mathbf{project}(\mathbf{x}))$$
$$f := gain_{0.9}(bias_{0.3}(s)) * D(\mathbf{x})$$

Figure **furry donut** shows this algorithm applied to a donut shaped object. The *bias* and *gain* adjustments are used to shape the noise profile into a relatively hard boundary (*gain* adjustment) with more empty space than hair (*bias* adjustment). The hairs will all be of the same length, equal to the width of the object's soft region. The projection function can be computed analytically here, since the object's geometry is well understood. This is an example of a geometry-dependent DMF.

To make the fur curly as in figure **tribble** we use *noise* to displace x before projecting. We use a vector valued function noise for this, instead of the usual scalar *noise*



furry donut



tribble

function, so that the displacement will occur in an arbitrary direction. This creates a sort of three dimensional ripple glass effect, as though straight hair were being seen through a distorted space:

$$\begin{aligned}
&d := D(\mathbf{x}) \\
&\mathbf{x}' := \mathbf{x} + gain_{0.8}(1-d) * curliness * \mathbf{noise}(\mathbf{x}) \\
&s := noise(freq * \mathbf{project}(\mathbf{x}')) \\
&f := gain_{0.9}(bias_{0.3}(s)) * d
\end{aligned}$$

There are several things to note in the above algorithm. The scalar variable *curliness* controls the magnitude of the curl. We use the expression $gain_{0.8}(1-d)$ to shape the curl, so that the hairs are initially straight where they grow out of the root (where $D = 1.0$), and gradually curl up as they reach the outer boundary (where $D = 0.0$). The vector valued **noise** function is built from *noise* as:

$$\mathbf{noise} = \left[noise(\mathbf{x}-\sigma), noise(\mathbf{x}), noise(\mathbf{x}+\sigma) \right]$$

where the offset vector σ is made large enough so that the three calls to *noise* are guaranteed to return uncorrelated values (since each call will encounter an entirely different set of pseudorandom knots).

Note how in all of the above we create a relatively simple algorithmic mechanism with a small number of scalar variables that control aspects of perceptual interest (filament length, fineness, curliness, etc). As in [5], these are essentially knobs that a user of a hypertexture system can adjust at a high level to synthesize a particular variety of fur, without necessarily knowing the details of furrier synthesis.

3. Rendering Hypertexture

Implementation of hypertexture rendering is only practical using volume rendering techniques, since hypertextural objects often have no well defined surfaces. Since volume rendering techniques have time complexity $O(n^3)$ with respect to resolution, they are typically slow. Fortunately the DMF evaluation is independent at each sample point, so hypertexture is particularly suitable for parallel or distributed implementations. We have found that designing and rendering hypertexture can be done on a serial von Neumann machine, but is *much* more enjoyable when computed in parallel. This section focuses on the hypertexture rendering algorithm and its different implementations.

3.1 Ray Marching Algorithm

In this section, we describe the *ray marching* algorithm [1] to generate images of hypertexture. As in traditional ray casting, the ray marcher casts a ray into model space for every pixel. We first clip each ray to a parallelpiped that bounds the hypertexture volume, using the optimal bounding test of [13]. If the ray does not intersect the parallelpiped, we move to the next pixel for processing. If the ray does intersect the parallelpiped, the ray parameters μ_0 and μ_1, representing respectively the entry and exit points of the parallelpiped, are computed. Ray marching begins at the ray parameter value μ_0, and proceeds at a fixed increment $\Delta\mu$. We sample the model along the ray at points:

$$\mathbf{x} = \mathbf{x}_{\mu_0} + k\ \Delta\mathbf{x}_{\mu}$$

where

$$k = 0,1,2,\ldots \text{ such that } \mu_0 + k\ \Delta\mu \le \mu_1$$

and $\Delta\mathbf{x}_{\mu}$ is the displacement along the ray, in model space, at each increment.

Aliasing of hypertexture is a potential problem. In practice we have achieved excellent results by manually tuning each hypertexture to be frequency clamped [17] as a function of ray-marching step size. For example, in the turbulence function described above, we stop the iteration when the period of the noise function is as small as the size of $\Delta\mathbf{x}_{\mu}$. The fact that an empirical approach works so well is encouraging but is clearly not definitive†.

At each point along the ray, we first evaluate a DMF $f(\mathbf{x})$. If $0 < f(\mathbf{x}) < 1$, then we evaluate the field gradient ∇f, normalize it, and use it as a normal vector for diffuse and/or specular shading, as well as for any refraction and reflection computations. To compute ∇f it is sufficient to evaluate f at two points perpendicularly off the ray in mutually perpendicular directions $\Delta\mathbf{x}_{\nu}$ and $\Delta\mathbf{x}_{\omega}$. Since we have already computed the result of f at our previous sample $\mathbf{x}-\Delta\mathbf{x}_{\mu}$, we can then approximate the gradient of f with respect to (μ,ν,ω) by the finite difference vector:

$$\left[f(\mathbf{x})-f(\mathbf{x}-\Delta\mathbf{x}_{\mu}), f(\mathbf{x}+\Delta\mathbf{x}_{\nu})-f(\mathbf{x}), f(\mathbf{x}+\Delta\mathbf{x}_{\omega})-f(\mathbf{x}) \right]$$

To convert this gradient vector in (μ,ν,ω) space into a gradient vector in (x,y,z) space, we multiply by the transformation matrix:

$$\begin{bmatrix} \Delta\mathbf{x}_{\mu} \\ \Delta\mathbf{x}_{\nu} \\ \Delta\mathbf{x}_{\omega} \end{bmatrix}$$

---

† A more systematic approach to antialiasing hypertexture is still a subject of research. We conjecture that it will involve discovering perceptually equivalent tradeoffs between different base functions (so that, for example, increasing gain at small scales, which increases gradient will automatically force the properly reduced amplitude).

The ray basis vectors $\Delta\mathbf{x}_\mu$, $\Delta\mathbf{x}_v$, and $\Delta\mathbf{x}_\omega$ can be precomputed once per ray. For an orthogonal view, they only need be computed once for the entire image.

Note that the expense of ray marching triples within an object's soft region, since computing ∇f requires us to evaluate f three times instead of just once per sample.

We must also accumulate opacity for visibility determination. A method similar to that of [7] is employed along the ray, so that at the *kth* step:

$$\begin{aligned} & t := \alpha_k(1-\alpha) \\ & color := color + t * color_k \\ & \alpha := \alpha + t \end{aligned}$$

where $color_k$ and α_k are sample color and opacity, respectively. To ensure resolution independence, we make opacity a function of both density and step-size:

$$\alpha_k := 1-(1-density)^{c\ *\ \text{step-size}}$$

where c is a normalizing constant. Color is the product of the shading value and the user specified color. The color mapping can be as simple or as complex as desired; step functions or splines may be appropriate. This same approach to color mapping was taken in [5]. Color maps are typically used to identify or isolate particular features in a scalar field [10]. Here again the local gradient ∇f is used to estimate the local normal vector. The use of opacity allows us to see amorphous or very fine volumetric features with a great deal of clarity.

The above differs from the method of [7], since we proceed in a front to back order (as we move along the ray from its entry point towards its exit point) as opposed to back to front order. If the accumulated opacity reaches unity, the evaluation along a ray stops before we exit the volume. This avoids unnecessary computation for regions that are entirely obscured.

Since in practice we have been able to tune our hypertextures empirically to be frequency clamped [17], do have not needed to use supersampling. We note though that stochastic ray marching might be employed as an extension of our algorithm, implemented by firing jittered rays with jittered step phases within each pixel. Similarly, motion blur might be achieved by sampling rays over time [21]. Other improvement might be made by utilizing recent work on optimization of ray marching [12].

Running Time

An $n \times n$ image requires $O(n^3)$ sample evaluations, so running time rises dramatically with increased resolution. Since increasing resolution by a factor of 4×4 increases running time by 64, we usually do our quick "is it even there?" tests at a resolution of 32×32, which take a few seconds, our rough tests at 128×128, which take a few minutes, and our final runs at 512×512, which take a few hours if run serially. As a rule of thumb, we see how many seconds a run takes at 32×32 to estimate how many serial hours it will be at 512×512. Final runs would take anywhere from 3 to 15 hours, depending on hypertexture complexity, on a single Sun 4-260 workstation. To improve this performance, we have taken the two following approaches to parallelizing the algorithm.

Parallel Rendering

The ray marcher was implemented in C on an AT&T Pixel Machine [14], an MIMD coarse-grain computer with general purpose floating-point processors, having relatively little memory per processor. In our implementation each of 64 processors uses an identical hypertexture program to compute a different interleaved subset of the final screen image, an arrangement that tends to optimize load balancing among processors.

Our memory requirements are particularly small because hypertexture is highly procedural. The only significant data space required is for color maps, alpha map and the noise function lookup tables. The sum of these is less than 5 Kbytes. Thus we are able to maintain a complete copy of the database at each processor.

Since hypertexture can be evaluated independently at each pixel, and because each processor maintains its own database, this implementation has the following characteristics:

- it executes independently and asynchronously at each node
- no interprocessor communication is required
- the cost associated with parallelizing the algorithm is negligible

We have found that increasing the number of processors produces a linear (optimal) decrease in execution time. Therefore in principle hypertexture could be generated interactively, since with enough processors, the time to render an image would be bounded only by the time to render the slowest pixel (in our experience, a fraction of a second at most).

Distributed Rendering

In addition to implementing the ray-marcher on a parallel computer, we also implemented the distributed ray-marching computation over a local-area network of Unix workstations on a shared Network File System. Each workstation runs the identical program, compiled for its

[17] Norton, A. Clamping: A Method of Antialiasing Textured Surfaces by Bandwidth Limiting in Object Space. In *Computer Graphics 16*, 3 (August 1982).

[18] Zadeh, L. A., *Fuzzy Sets and Applications (selected papers)*, John Wiley and Sons, New York, 1987.

[19] Zimmerman, H. J., *Fuzzy Set Theory - and Its Applications*, Kluwer-Nijhoff, Hingham, 1985, pp. 30-36.

[20] Menzel, D. H., ed., *Fundamental Formulas of Physics, vol. 2*, Dover, New York, 1960, pp. 370-371.

[21] Cook, R., Distributed Ray Tracing, In *Computer Graphics 18*, 3 (August 1984).

[22] Perlin, K., *Synthesizing Realistic Textures by the Composition of Perceptually Motivated Functions* [Ph.D. Dissertation], New York University, (Feb. 1986).

[23] Kajiya, J., Anisotropic Reflection Models. In *Computer Graphics 19*, 3 (August 1985).

particular processor, to compute different pixels of the same image. In practice we have observed a linear speedup over the single processor serial version of the algorithm, using the dozen or so workstations in our lab.

4. Summary, Conclusions and Future

We have described a new modeling technique which modulates shape by applying procedural texture to a continuous volumetric region. The method contrasts with previous techniques in that we manipulate matter throughout R^3, instead of only at surfaces. This approach allows us to create the appearance of complex, real-world phenomena that would be difficult or impossible to generate with previous methods. The computational model is $O(n^3)$ but optimally parallelizable, achieving linear decreases in execution time with increases in the number of processors.

Clearly the model as described is highly empirical, leaving unanswered the disturbing question of why such simple techniques produce such visually convincing results. Prior work [22] has led us to believe that there is a sound perceptual basis for this, and that in general procedural textures can be organized into a human perceptual taxonomy. We plan in future work to extend this taxonomy to the description of hypertexture.

Our newest research concentrates on applying hypertexture to empirical shape data such as cranio-facial structures and teapots. By performing preprocessing passes through volumetric shape images, we are currently implementing cast shadows and extending geometry-dependent functions such as the **project** operator of fur hypertexture to empirical shapes at $O(n^3)$ cost. We also plan to incorporate more sophisticated shading models, in particular the anisotropic shading of Kajiya [23].

5. Acknowledgements

The authors greatly appreciate the insightful comments of Don Mitchell, Jim Conant, Dave Weimer, Jim Demmel, and Mark Perlin. The reviewers' comments were invaluable. In particular Ken would like to honor the request of one reviewer by offering a most humble apology for inflicting the term "furrier synthesis" on an unsuspecting scientific populace.

References

[1] Tuy, H. and Tuy, L. Direct 2-D Display of 3-D Objects, *IEEE Computer Graphics and Applications 4*, 10 (October 1984), pp. 29-33.

[2] Lorensen, W. Marching Cubes: A High Resolution 3D Surface Construction Algorithm, In *Computer Graphics 21*, 4 (July 1987), pp. 163-169.

[3] Perlin, K. Functionally Based Modeling. *SIGGRAPH Course Notes* (August 1988).

[4] Frieder, G., Gordon, D. and Reynolds, R. A. Back-to-Front Display of Voxel-Based Objects, *IEEE Computer Graphics and Applications*, (January 1985), pp. 52-60.

[5] Perlin, K. An Image Synthesizer, In *Computer Graphics 19*, 3 (July 1985).

[6] Kaufman, Arie. Efficient Algorithms for 3D Scan-Conversion of Parametric Curves, Surfaces and Volumes, In *Computer Graphics 21*, 4 (July 1987).

[7] Levoy, Marc. Volume Rendering: Display of Surface from Volume Data, *IEEE Computer Graphics and Applications* (May 1988), pp. 29-36.

[8] Drebin, R., Carpenter, L. and Hanrahan, P. Volume Rendering, In *Computer Graphics 22*, 4 (August 1988).

[9] Sabella, Paolo. A Rendering Algorithm for Visualizing 3D Scalar Fields, In *Computer Graphics 22*, 4 (August 1988).

[10] Upson, C. and Keeler, M. V-BUFFER: Visible Volume Rendering, In *Computer Graphics 22*, 4 (August 1988).

[11] Kaufman, A. and Bakalash, R. A 3D Cellular Frame Buffer, *Proceedings of EUROGRAPHICS 1985* (September 1985), Nice, France, pp. 215-220.

[12] Amanatides, J. and Woo, A., A Fast Voxel Traversal Algorithm for Ray Tracing, *Proceedings of EUROGRAPHICS 1987* (Amsterdam, Holland), pp. 3-10.

[13] Toth, D. L., On Ray Tracing Parametric Surfaces, In *Computer Graphics 19*, 3 (August 1985).

[14] Potmesil, M. and Hoffert, E., The Pixel Machine: A Parallel Image Computer, In *Computer Graphics 23*, 3 (August 1989).

[15] Blinn, J., A Generalization of Algebraic Surface Drawing, *"ACM Transactions on Graphics 1,"* pp 235., 1982.

[16] Kajiya, J., Herzen, B., "Ray Tracing Volume Densities," In *Computer Graphics 18*, 3 (August 1984).