12

NOISE, HYPERTEXTURE, ANTIALIASING, AND GESTURE

KEN PERLIN

INTRODUCTION

This first section touches on several topics that relate to the work I've done in procedural modeling. A brief introduction to hypertexture and a review of its fundamental concepts is followed by a discussion of the essential functions needed in order to be able to easily tweak procedural models.

The second section is a short tutorial on how the *noise* function is constructed, followed by a discussion of how raymarching for hypertexture works and some examples of hypertexture. Next comes an interesting possible approach for antialiasing procedurally defined images. Then we discuss a surface representation based on sparse wavelets. We conclude by applying notions from procedural modeling to human figure motion.

Shape, Solid Texture, and Hypertexture

The precursor to hypertexture was *solid texture.* Solid texturing is simply the process of evaluating a function over R^3 at each visible surface point of a rendered computer graphic (CG) model. The function over R^3 then becomes a sort of solid material, out of which the CG model shape appears to be "carved."

I became interested in what happens when you start extending these texture functions off of the CG surface. What do they look like as space-filling functions? So I developed a simple formalism that allowed me to play with this idea, by extending the notion of *characteristic function.*

Traditionally, the shape of any object in a computer graphic simulation is described by a characteristic function—a mapping from each point in R^3 to the

Boolean values **true** (for points inside the object) or **false** (for points outside the object). This is a point set. The boundary of this point set is the surface of the object.

I replaced the Boolean characteristic function with a continuous one. We define for any object a characteristic function that is a mapping from f: $R^3 \rightarrow [0 \ldots 1]$. All points $\vec{x}$ for which $f(\vec{x})$ is 0 are said to be *outside* the object. All points $\vec{x}$ for which $f(\vec{x})$ is 1 are said to be *strictly inside* the object. Finally, all points $\vec{x}$ for which $0 < f(\vec{x}) < 1$ are said to be in the object's *fuzzy region.*

This formulation gives the object surface an appreciable thickness. We can now combine solid texture functions with the function that describes the object's fuzzy region. In this way shape and solid texture become unified—a shape can be seen as just a particular solid texture function. I refer to this flavor of texture modeling as *hypertexture,* since we are texturing in a higher dimension (the full three-dimensional object space)—and also because it's a really atrocious pun.

TWO BASIC PARADIGMS

There are two distinct ways that solid textures can be combined with fuzzy shapes. You can either use the texture to distort the space before evaluating the shape, or add the texture value to a fairly soft shape function, then apply a "sharpening" function to the result.

The first approach involves calculations of the form

$$f(\vec{x}) = \text{shape}(\vec{x} + \text{texture}(\vec{x})\vec{v})$$

where $\vec{v}$ is a simple vector expression. The second approach involves calculations of the form

$$f(\vec{x}) = \text{sharpen}(\textit{shape}(\vec{x}) + \text{texture}(\vec{x}))$$

Bias, Gain, and So Forth

The secret to making good procedural textures is an interface that allows you to tune things in a way that makes sense to you. For example, let's say that there is some region over which a function value is varying from 0.0 to 1.0. Perhaps you realize that this function should be "pushed up" so that it takes on higher values in its middle range. Or maybe you want to push its values a bit toward its high and low values and away from the middle.

You could do these things with spline functions or with power functions. For example, let's say that $f(t)$: $R \rightarrow [0 \ldots 1]$. Then $pow(f(t), 0.8)$ will push the values of f up toward 1.0. But this is not intuitive to use because it's hard to figure out what values to put into the second argument to produce a particular visual result. For this reason, I've built two functions that I use constantly, just to do these little continuous tweaks.

Bias

To get the same functionality provided by the power function, but with a more intuitive interface, I've defined the function $bias_b$, which is a power curve defined over the unit interval such that $bias_b\,(0.0) = 0.0$, $bias_b\,(0.5) = b$, and $bias_b\,(1.0) = 1.0$. By increasing or decreasing b, we can thus bias the values in an object's fuzzy region up or down. Note that $bias_{0.5}$ is the identity function.

Bias is defined by

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

Gain

Similarly, we often want an intuitive way to control whether a function spends most of its time near its middle range or, conversely, near its extremes. It's sort of like having a gain on your TV set. You can force all the intensities toward middle gray (low gain) or, conversely, force a rapid transition from black to white (high gain).

We want to define $gain_g$ over the unit interval such that

$$\begin{aligned} gain_g\,(0.0) &= 0.0 \\ gain_g\,(0.25) &= 0.5 - g/2 \\ gain_g\,(0.5) &= 0.5 \\ gain_g\,(0.75) &= 0.5 + g/2 \\ gain_g\,(1.0) &= 1.0 \end{aligned}$$

By increasing or decreasing g, we can thus increase or decrease the rate at which the midrange of an object's fuzzy region goes from 0.0 to 1.0. Note that $gain_{0.5}$ is the identity function.

Motivated by the above, I've defined gain by

$$\text{if } t > 0.5 \text{ then } \frac{bias_{1-g}(2t)}{2} \text{ else } \frac{2 - bias_{1-g}(2 - 2t)}{2}$$

The above two functions provide a surprisingly complete set of tools for tweaking things. If you want to make something look "darker" or "more transparent," you would generally use $bias_b$ with $b < 0.5$. Conversely, if you want to make something look "brighter" or "more opaque," you would generally use $bias_b$ with $b > 0.5$.

Similarly, if you want to "fuzz out" a texture, you would generally use $gain_g$ with $g < 0.5$. Conversely, if you want to "sharpen" a texture, you would generally use $gain_g$ with $g > 0.5$. Most of the time, you'll want to instrument various numerical parameters with bias and gain settings and tune things to taste.

CONSTRUCTING THE *NOISE* FUNCTION

Part of the reason that procedural texture is effective is that it incorporates randomness in a controlled way. Most of the work involved in achieving this is contained inside the *noise* function, which is an approximation to what you would get if you took white noise (say, every point in some sampling mapped to a random value between -1.0 and 1.0) and blurred it to dampen out frequencies beyond some cutoff.

The key observation here is that, even though you are creating "random" things, you can use *noise* to introduce high frequencies in a controlled way by applying *noise* to an appropriately scaled domain. For example, if $\vec{x}$ is a point in R^3, then *noise* $(2\vec{x})$ introduces frequencies twice as high as does *noise* $(\vec{x})$. Another way of saying this is that it introduces details that are twice as small.

Ideally the *noise* function would be constructed by blurring white noise, preferably by convolving with some Gaussian kernel. Unfortunately, this approach would necessitate building a volume of white noise and then blurring it all at once. This is quite impractical.

Instead, we want an approximation that can be evaluated at arbitrary points, without having to precompute anything over some big chunk of volume. The approach described here is the original one I came up with in 1983, which I still use. It's a kind of spline function with pseudorandom knots.

The key to understanding the algorithm is to think of space as being divided into a regular lattice of cubical cels, with one pseudorandom wavelet per lattice point.

You shouldn't get scared off by the use of the term "wavelet." A wavelet is simply a function that drops off to zero outside of some region and that integrates to zero. This latter condition means that the wavelet function has some positive region and some negative region, and that the two regions balance each other out. When I talk about the "radius" of a wavelet, I just mean the distance in its domain space from the wavelet center to where its value drops off to zero.

We will use wavelets that have a radius of one cel. Therefore, any given point in R^3 will be influenced by eight overlapping wavelets—one for each corner of the cel containing the point.

Computation proceeds in three successive steps:

- Compute which cubical "cel" we're in.
- Compute the wavelet centered on each of eight vertices.
- Sum the wavelets.

Computing Which Cubical "Cel" We're In

The "lowest" corner of the cel containing point $[x, y, z]$ is given by

$$[i, j, k] = [\lfloor x \rfloor, \lfloor y \rfloor, \lfloor z \rfloor]$$

Then the eight vertices of the cel containing $[x, y, z]$ are

$$\begin{aligned}
&(1)\ [i, j, k] \\
&(2)\ [i + 1, j, k] \\
&(3)\ [i, j + 1, k] \\
&\quad \ldots \\
&(8)\ [i + 1, j + 1, k + 1]
\end{aligned}$$

Finding the Pseudorandom Wavelet at Each Vertex of the Cel

For each of the above lattice points, we want to find what wavelet to apply. Ideally, the mapping from cels to wavelet coefficients would be nonrepeating. It turns out, though, that because *noise* has no large-scale structure, the mapping can eventually repeat itself, without this repetition being at all noticeable. This allows us to use a relatively simple table lookup scheme to map vertices to coefficients. I find that a repeat distance of 256 is more than large enough.

The basic idea is to map any $[i, j, k]$ into a unique number between 0 and 255. We precompute a table G of 256 pseudorandom coefficient values and always index into this fixed set of values.

In the following two sections we will accomplish two things. First, we will discuss what properties we want from the wavelet coefficients and how to precompute G—a table of 256 sets of coefficients that has these properties. Then we will discuss the properties we want in the mapping from $P : [i, j, k] \rightarrow 0 \ldots 255$ and how to achieve these properties.

Wavelet Coefficients

I actually force the *noise* function to take on a value of zero at each lattice point. This ensures that it can't have any appreciable energy at low spatial frequencies. To achieve this, I give a value at its center of zero to each of the overlapping wavelets that are being summed and then give each wavelet a radius of one—so that each wavelet will reach a value of zero just as it gets to the center of its neighbors.

This means that we can define the wavelet centered at each lattice point as a smooth function with the following properties:

- It has a value of zero at its center.
- It has some randomly chosen gradient at its center.
- It smoothly drops off to zero a unit distance from its center.

So we have to randomly choose a gradient for each wavelet center. I do this via a Monte Carlo method—precompute a table of gradients in uniformly random directions and then index into this table. To create the gradient table, I need a set of points that are uniformly distributed on the surface of a unit sphere. I ensure that this set of points will be uniformly distributed as follows:

- Choose points uniformly within the cube $[-1 \ldots 1]^3$.
- Throw out any points falling outside of the unit sphere.
- Project surviving points onto the unit sphere.

The pseudocode to do this is

```
for i in [0 . . . 255]
  repeat
    x = random(−1. · · · +1.)
```

$y = random(-1. \cdots +1.)$
$z = random(-1. \cdots +1.)$
until $x^2 + y^2 + z^2 < 1.0$
$G[i] = normalize\ [x, y, z]$

To Quickly Index into *G* in a Nonbiased Way

Now we need to find a way to answer the following question: If any point in R^3 is in the vicinity of a wavelet, how can it rapidly get the coefficients of that wavelet?

We basically want a really random way to map lattice points $[i, j, k]$ to indices of G. We have to avoid regularities in this mapping, since any regularity would be extremely visible to somebody looking at the final *noise* function.

I use the following method:

- Precompute a "random" permutation table P.
- Use this table to "fold" $[i, j, k]$ into a single n.

In this section I describe first how to create a suitable permutation table and then how to use this table to "fold" $[i, j, k]$ into the range $[0 \ldots 255]$.

The permutation table can be precomputed, so that step doesn't need to be especially fast. The pseudorandom permutation table P is created by

for i in $[0 \ldots 255]$
  $j = random[0 \ldots 255]$
  exchange $P[i]$ *with* $P[j]$

The folding function $fold(i, j, k)$ is then computed by

$$n = P[i_{\bmod 256}]$$
$$n = P[(n + j)_{\bmod 256}]$$
$$n = P[(n + k)_{\bmod 256}]$$

For added speed, I don't actually do the mod operations. Instead, I precompute P to be twice as long, setting $P[256 \ldots 511] := P[0 \ldots 255]$. Then if $0 \le i, j, k \le 255$, we can just do $P[P[P[i] + j] + k]$ for each wavelet.

Now for each vertex of the unit cube whose "lowest" corner is $[i, j, k]$, we can quickly obtain wavelet coefficients as follows:

(1) $G(fold(i, j, k))$
(2) $G(fold(i + 1, j, k))$

$$(3)\ G(fold(i, j + 1, k))$$
$$\ldots$$
$$(8)\ G(fold(i + 1, j + 1, k + 1))$$

Evaluating the Wavelet Centered at [i, j, k]

The remaining steps to finding $noise(x, y, z)$ are now as follows:

1. Each wavelet is a product of
   - a cubic weight that drops to zero at radius 1
   - a linear function, which is zero at (i, j, k)
2. To compute the wavelet we must
   - get (x, y, z) relative to the wavelet center
   - compute the weight function
   - multiply the weight by the linear function

   First we must get (x, y, z) relative to the wavelet center:

$$[u, v, w] = [x - i, y - j, z - k]$$

Note that u, v, w values are bounded by $-1 \le u, v, w \le 1$.

Now we must compute the dropoff $\Omega_{(i,j,k)}(u, v, w)$ about $[i, j, k]$:

$$\Omega_{(i,j,k)}(u, v, w) = drop(u) \times drop(v) \times drop(w)$$

where each component dropoff is given by the cubic approximation

$$\text{drop}(t) = 1. - 3|t|^2 + 2|t|^3 \qquad \text{(but zero whenever } |t| > 1)$$

and we must multiply this by the linear function that has the desired gradient and a value of zero at the wavelet center:

$$G_{(i,j,k,)} \cdot [u, v, w]$$

The value of $\text{wavelet}_{(i,j,k)}$ at (x, y, z) is now given by

$$\Omega_{(i,j,k)}(u, v, w)(G_{(i,j,k)} \cdot [u, v, w])$$

Finally, $noise\ (x, y, z)$ is given by the sum of the eight wavelets near (x, y, z).

Following is a complete implementation of *noise* over R^3:

```
/* noise function over R3-implemented by a pseudorandom tricubic spline */

#include <stdio.h>
#include <math.h>
```

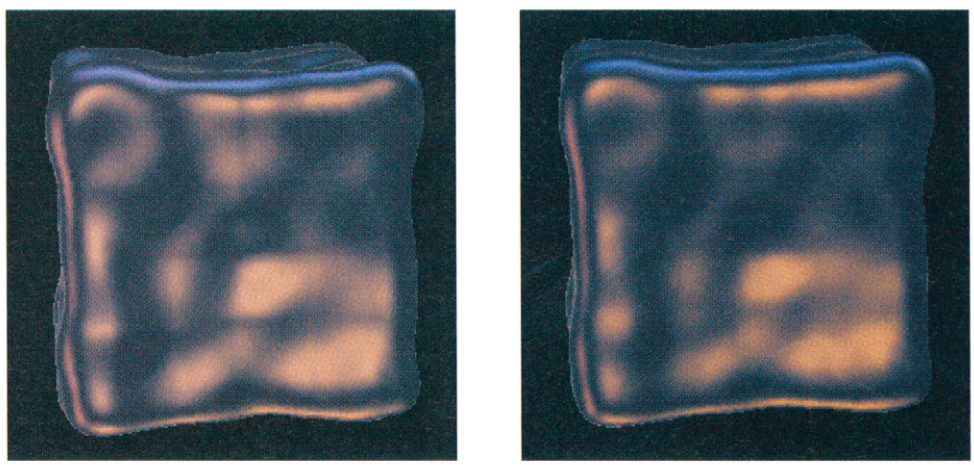
```
#define DOT(a,b) (a[0] * b[0] + a[1] * b[1] + a[2] * b[2])

#define B 256

static p[B + B + 2];
static float g[B + B + 2][3];
static start = 1;

#define setup(i,b0,b1,r0,r1) \
     t = vec[i] + 10000.; \
     b0 = ((int)t) & (B-1); \
     b1 = (b0+1) & (B-1); \
     r0 = t - (int)t; \
     r1 = r0 - 1.;

float noise3(vec)
float vec[3];
{
     int bx0, bx1, by0, by1, bz0, bz1, b00, b10, b01, b11;
     float rx0, rx1, ry0, ry1, rz0, rz1, *q, sy, sz, a, b, c, d, t, u, v;
     register i, j;

     if (start) {
          start = 0;
          init();
     }

     setup(0, bx0,bx1, rx0,rx1);
     setup(1, by0,by1, ry0,ry1);
     setup(2, bz0,bz1, rz0,rz1);

     i = p[ bx0 ];
     j = p[ bx1 ];

     b00 = p[ i + by0 ];
     b10 = p[ j + by0 ];
     b01 = p[ i + by1 ];
     b11 = p[ j + by1 ];

#define at(rx,ry,rz) ( rx * q[0] + ry * q[1] + rz * q[2] )

#define s_curve(t) ( t * t * (3. - 2. * t) )

#define lerp(t, a, b) ( a + t * (b - a) )

     sx = s_curve(rx0);
     sy = s_curve(ry0);
     sz = s_curve(rz0);
```

```
    q = g[ b00 + bz0 ] ; u = at(rx0,ry0,rz0);
    q = g[ b10 + bz0 ] ; v = at(rx1,ry0,rz0);
    a = lerp(sx, u, v);

    q = g[ b01 + bz0 ] ; u = at(rx0,ry1,rz0);
    q = g[ b11 + bz0 ] ; v = at(rx1,ry1,rz0);
    b = lerp(sx, u, v);

    c = lerp(sy, a, b);          /* interpolate in y at low x */

    q = g[ b00 + bz1 ] ; u = at(rx0,ry0,rz1);
    q = g[ b10 + bz1 ] ; v = at(rx1,ry0,rz1);
    a = lerp(sx, u, v);

    q = g[ b01 + bz1 ] ; u = at(rx0,ry1,rz1);
    q = g[ b11 + bz1 ] ; v = at(rx1,ry1,rz1);
    b = lerp(sx, u, v);

    d = lerp(sy, a, b);          /* interpolate in y at high x */

    return 1.5 * lerp(sz, c, d); /* interpolate in z */
}

static init()
{
    long random();
    int i, j, k;
    float v[3], s;

    /* Create an array of random gradient vectors uniformly on the
    unit sphere */
    srandom(1);
    for (i = 0 ; i < B ; i++) {
        do {                     /* Choose uniformly in a cube */
            for (j=0 ; j<3 ; j++)
                v[j] = (float)((random() % (B + B)) - B)/B;
                s = DOT(v,v);
        } while (s > 1.0);       /* If not in sphere try again */
        s = sqrt(s);
        for (j = 0 ; j < 3 ; j++)      /* Else normalize */
            g[i][j] = v[j] / s;
    }

    /* Create a pseudorandom permutation of [1 .. B] */
    for (i = 0 ; i < B ; i++)
        p[i] = i;
    for (i = B ; i > 0 ; i -= 2) {
        k = p[i];
        p[i] = p[j = random() % B];
        p[j] = k;
```

```
    }

    /* Extend g and p arrays to allow for faster indexing, */
    for (i = 0 ; i < B + 2 ; i++) {
        p[B + i] = p[i];
        for (j = 0 ; j < 3 ; j++)
        g[B + i][j] = g[i][j];
    }
}
```

RECENT IMPROVEMENTS TO THE *NOISE* FUNCTION

I recently made some tweaks to the *noise* function that make it look better and run somewhat faster (Perlin 2002). Note that the cubic interpolation polynomial $3t^2 - 2t^3$ has a derivative of $6 - 12t$, which has nonzero first and second derivatives at $t = 0$ and $t = 1$. These create second-order discontinuities across the coordinate-aligned faces of adjoining cubic cells, which become noticeable when a noise-displaced surface is shaded, and the surface normal (which is itself a derivative operator) acquires a visibly discontinuous derivative. We can instead use the interpolation polynomial $6t^5 - 15t^4 + 10t^3$, which has zero first and second derivatives at $t = 0$ and $t = 1$. The improvement can be seen by comparing the two noise-displaced superquadrics (see Figure 12.1).



(a) (b)

FIGURE 12.1 (a) Noise interpolated with $3t^2 - 2t^3$; (b) noise interpolated with $6t^5 - 15t^4 + 10t^3$.

Also, we can speed up the algorithm while reducing grid-oriented artifacts, by replacing the *g* array with a small set of fixed gradient directions:

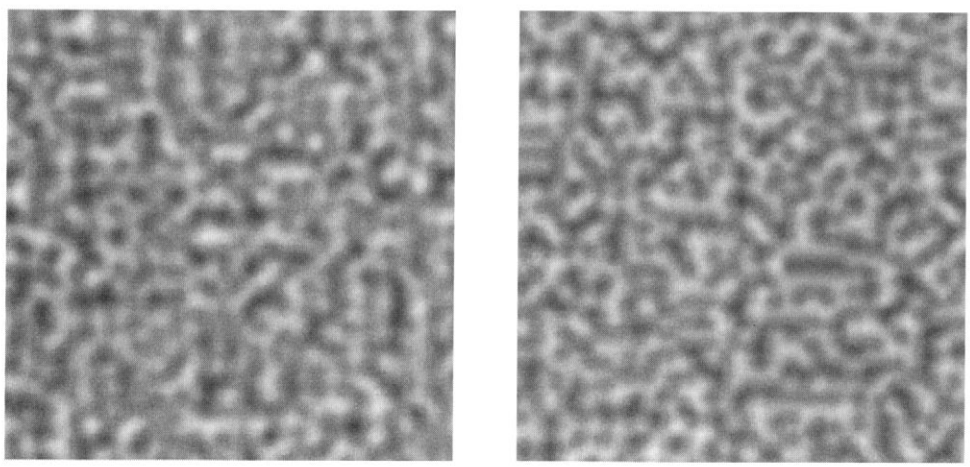
$$(1, 1, 0),(-1, 1, 0),(1, -1, 0),(-1, -1, 0),$$
$$(1, 0, 1),(-1, 0, 1),(1, 0, -1),(-1, 0, -1),$$
$$(0, 1, 1),(0, -1, 1),(0, 1, -1),(0, -1, -1)$$

This set of gradient directions does two things: (1) it avoids the main axis and long diagonal directions, thereby avoiding the possibility of axis-aligned clumping, and (2) it allows the eight inner products to be effected without requiring any multiplies, thereby removing 24 multiplies from the computation. The improvement can be seen by comparing the two gradient strategies (see Figure 12.2).

RAYMARCHING

To see hypertexture, you need a *raymarcher* renderer. I've implemented the raymarcher in two parts. First I built a system layer—the part that never changes. This renders hypertexture by marching a step at a time along each ray, accumulating density at each sample along the way. There are all kinds of hooks for user interaction built into the raymarcher.

Then, for each type of hypertexture, I construct different "application" code, describing a particular hypertexture. At this point in the process I can safely wear the



(a) (b)

FIGURE 12.2 (a) Noise with old gradient distribution; (b) noise with new gradient distribution.

hat of a "naive user." Everything is structured so that at this level I don't have to worry about any system details. All I really need to worry about is what density to return at any point in R^3. This makes it very painless to try out different types of hypertexture.

System Code: The Raymarcher

I render hypertexture by "raymarching"—stepping front to back along each ray from the eye until either total opacity is reached or the ray hits a back clipping plane. Conceptually, the raymarching is done inside the unit cube: $-0.5 < x, y, z < 0.5$. A 4×4 viewing matrix transforms this cube to the desired view volume.

One ray is fired per pixel; the general procedure is as follows:

```
step = 1.0/resolution;
for (y = -0.5 ; y < 0.5 ; y += step)
for (x = -0.5 ; x < 0.5 ; x += step) {
    [point, point_step] = create_ray([x,y,-0.5], [x,y,-0.5+step], view_matrix);
    previous_density = 0.;
    init_density_function();                /* User supplied */
    color = [0,0,0,0];
    for (z = -0.5 ; z < 0.5 && color.alpha < 0.999 ; z += step} {

         density = density_function(point); /* User supplied */
         c = compute_color(density);        /* User supplied */

    /* Do shading only if needed */

        if (is_shaded && density != previous_density) {
            normal = compute_normal(point, density);
            c = compute_shading(c, point, normal); /* User supplied */
            previous_density = density;
        }

    /* Attenuation varies with resolution */

        c[3] = 1.0 - pow( 1.0 - c[3], 100. * step );

    /* Integrate front to back */

        if (c[3] > 0.)} {
            t = c[3] * (1.0 - color.alpha);
            color += [ t*c.red, t*c.green, t*c.blue, t ];
             }
```

```
    /* March further along the ray */

       point += point_step;
    }
}
```

Application Code: User-Defined Functions

The "user" gets to define four functions to create a particular hypertexture:

```
void init_density_function();
float density_function(float x, float y, float z);
color compute_color(float density);
color compute_shading(color c, vector point, vector normal);
```

What makes things really simple is that as a user you only have to define behavior at any given single point in space—the raymarcher then does the rest for you.

- `init_density_function()`—This function is called once per ray. It gives you a convenient place to compute things that don't change at every sample.
- `density_function()`—This is where you specify the mapping from points to densities. Most of the behavior of the hypertexture is contained in this function.
- `compute_color()`—Here you map densities to colors. This also gives you a chance to calculate a refractive index.
- `compute_shading()`—Nonluminous hypertextures react to light and must be shaded. The model I use is to treat any substance that has a density gradient as a translucent surface, with the gradient direction acting as a normal vector, as though the substance consists of small, shiny, suspended spheres.

In the raymarcher library I've included a Phong shading routine. I usually just call that with the desired light direction, highlight power, and so on.

Shading is relatively expensive, since it requires a normal calculation. Also, in many cases (e.g., self-luminous gases) shading is not necessary. For this reason, shading is only done if the user sets an `is_shaded` flag.

The raymarcher computes normals for shading by calling the user's density function three extra times:

```
vector = compute_normal(point, density) {
   vector d = = [
       density_function[point.x - epsilon, point.y, point.z] - density,
       density_function[point.x, point.y - epsilon, point.z] - density,
       density_function[point.x, point.y, point.z - epsilon] - density ];
   return d / |d|;
}
```

The preceding is the basic raymarcher. Two features have not been shown—refraction and shadows. Shadows are done by shooting secondary rays at each ray step where density != 0. They are prohibitively expensive except for hypertextures with "hard," fairly sharpened surfaces. In this case the accumulated opacity reaches totality in only a few steps, and so relatively few shadow rays need be followed.

Refraction is done by adding a fifth component to the color vector—an index of refraction. The user sets `c.irefract` (usually from density) in the `compute_color` function. The raymarcher then uses Snell's law to shift the direction of `point_step` whenever `c.irefract` changes from one step along the ray to the next. An example of this is shown in Figure 12.3.

Since the density can change from one sample point to the next, it follows that the normal vector can also change continuously. This means that refraction can occur continuously. In other words, light can travel in curved paths inside a

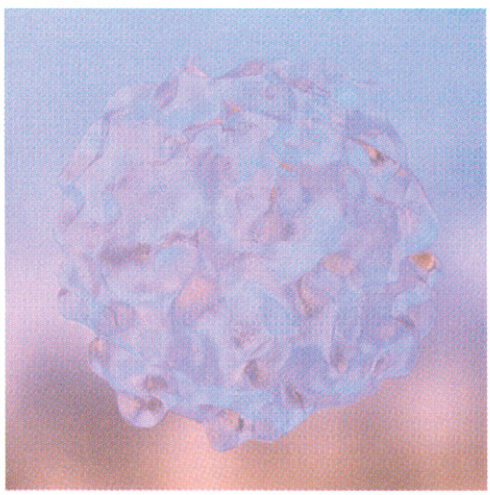


FIGURE 12.3 Blue glass.

hypertexture. This raises some interesting possibilities. For example, imagine a manufacturing process that creates wafers whose index of refraction varies linearly from one face to the other (probably by some diffusion process). By carving such a material, you could create optical components within which light travels in curved paths. It might be possible to do things this way that would be very difficult or impossible to do with traditional optical components (in which light only bends at discrete surfaces between materials). The results of such components should be quite straightforward to visualize using refractive hypertexture.

INTERACTION

Various kinds of interaction with hypertextures are possible, including modification of parameters and algorithmic models and various types of previewing, such as *z-slicing*.

Levels of Editing: Changing Algorithms to Tweaking Knobs

There are three levels of changes you can make. I describe them in order of slowest to implement (and most sweeping in effect) to fastest to implement.

- *Semantic changes—changing the user functions:* This is redefining your hypertexture methods. This type of change is covered in detail in the next section.
- *Parameters you change by editing an input file of numeric parameters:* This saves the time of recompiling the user functions, when all you want to change is some numeric value. The raymarcher has a mechanism built into it that lets you refer to a file that binds symbols to floating-point values when you run the program. These bindings are made accessible to the hypertexture designer at the inner rendering loop.

  In the following examples, I will adopt the following convention: Any symbol that begins with a capital letter refers to a parameter whose value has been set in this input file. Symbols beginning with lowercase letters refer to variables that are computed within the individual rays and samples.
- *Parameters you change from the command line:* These override any parameters with the same name in the input file. They are used to make animations showing things changing. For example, let's say you want to create an animation of a sphere with an expanding `Radius`, and you are working in the UNIX `csh` shell:

```
set i = 0
while ($i < 100)
   rm hypertexture -Radius $i sphere > $i
   @ i++
end
```

There are also some special parameters: `XFORM` for the view matrix, `RES` for image resolution, and `CLIP` for image clipping (when you just want to recalculate part of an image). These can be set either from the command line or as an environment variable (the former overrides the latter, of course).

In this chapter, I have hardwired numerical parameters into a number of expressions. These are just there to "tune" the model in various useful ways. For example, the expression "`100 * step`" appearing above in the attenuation step of the raymarcher has the effect of scaling the integrated density so that the user can get good results by specifying densities in the convenient range [0.0 . . . 1.0].

z-Slicing

For much of the time when designing hypertextures, you just need a general sense of the shape and position of the textured object. In this case it is useful to evaluate only at a fixed z—setting the value of a ray to the density at only one sample point a fixed distance away. This obviously runs many times faster than a full raymarch. I use z-slicing for general sizing and placement, often going through many fast iterations in z-slice mode to get those things just right.

SOME SIMPLE SHAPES TO PLAY WITH

Generally, the creation of a hypertexture begins with an algorithmically defined shape. This is the shape that will subsequently be modified to create the final highly detailed hypertexture.

Sphere

Start with a sphere with `inner_radius` and `outer_radius` defined. Inside `inner_radius`, density is everywhere 1.0. Beyond `outer_radius`, density has dropped completely to 0.0. The interesting part is the hollow shell in between:

```
/* (1) Precompute (only once) */

    rr0 = outer_radius * outer_radius;
    rr1 = inner_radius * inner_radius;
```

```
/* (2) radius squared */

    t = x * x + y * y + z * z;

/* (3) compute dropoff */

    if (t > rr0)
        return 0.;
    else if (t < rr1)
        return 1.;
    else
        return (t - rr0) / (rr1 - rr0);
```

Egg

To create an egg, you start with a sphere, but distort it by making it narrower at the top. A good maximal "narrowing" value is 2/3, which is obtained by inserting the following step into the sphere procedure:

```
/* (1.5) introduce eccentricity */

    e = ( 5. - y / outer_radius ) / 6.;
    x = x / e;
    z = z / e;
```

Notice that we must divide, not multiply, by the scale factor. This is because x and z are the arguments to a shape-defining function—to make the egg thinner at the top, we must increase (not decrease) the scale of x and z.

EXAMPLES OF HYPERTEXTURE

We now show various examples of hypertexture. Each example will illustrate one or more hypertexture design principles.

Explosions

The texture component here is turbulence uniformly positioned throughout space.

```
t = 0.5 + Ampl * turbulence(x, y, z);
return max(0., min(1. t));
```

Shape is just a sphere with `inner_radius` = 0.0, which ensures that the fuzzy region will consist of the entire sphere interior.

The density function is

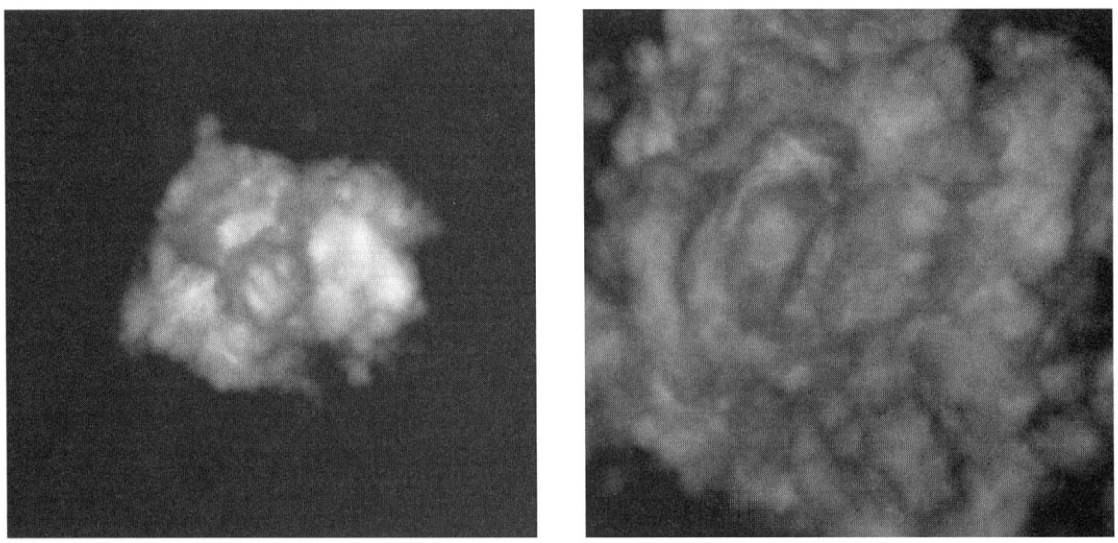
```
d = shape(x, y, z);
if (d > 0.)
    d = d * texture(x, y, z);
return d;
```

You can animate an explosion by increasing the sphere `outer_radius` over time. Figure 12.4(a) shows an explosion with `outer_radius` set to 0.4. Figure 12.4(b) shows the same explosion with `outer_radius` set to 0.8.

To create these explosions I oriented the cusps of the texture inward, creating the effect of locally expanding balls of flame on the surface. Contrast this with Figure 12.5 (Perlin and Hoffert 1989), where the cusps were oriented outward to create a licking flame effect.

Life-Forms

Just for fun, I placed a shape similar to the above explosions inside of an egg shape of constant density, as in Figure 12.6. By pulsing the `outer_radius` and `Ampl`



(a) (b)

FIGURE 12.4 (a) Explosion with `outer_radius` set to 0.4; (b) same explosion with `outer_radius` set to 0.8.

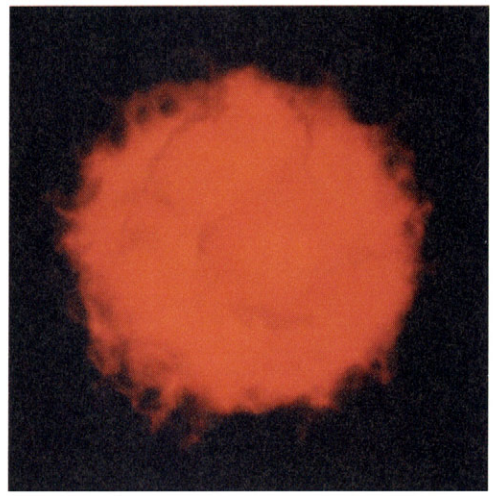


FIGURE 12.5 Fireball.

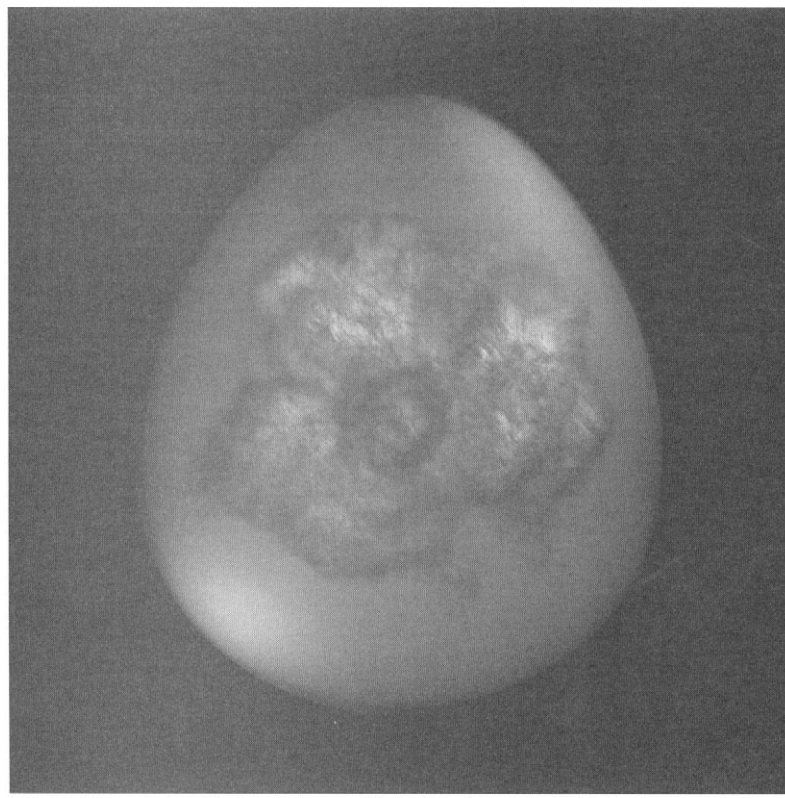


FIGURE 12.6 Explosion shape inside an egg.

rhythmically, while rotating slightly over time, I managed to hatch some rather intriguing simulations.

Space-Filling Fractals

Figure 12.7(a) through 12.7(d) shows steps in the simulation of a sparsely fractal material. At each step, *noise*() is used to carve volume away from the egg. Then *noise*() of twice the frequency is carved away from the remainder, and so on.

Figure 12.8 shows one possible result of such a process, a shape having infinite surface area and zero volume.

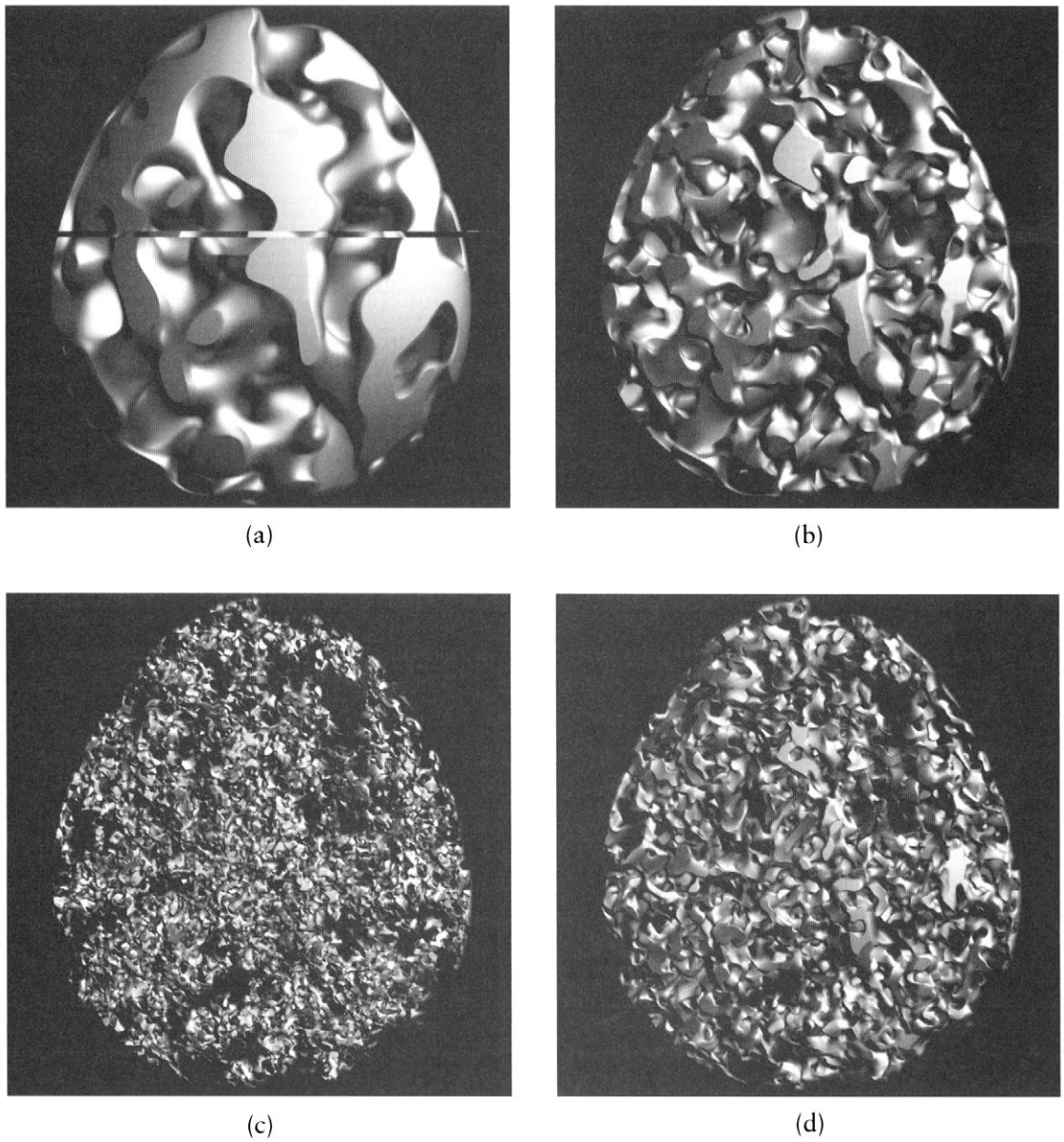
Woven Cloth

Cloth is defined by the perpendicular interweaving of warp threads and woof threads. We define a warp function `warp(x, y, z)`, where `y` is the direction perpendicular to the cloth:

```
/* (1) make an undulating slab */

   if (fabs(y) > PI)
       return 0.;

   y = y + PI/2 * cos(x) * cos(z);
   if (fabs(y) > PI/2)
       return 0.;

   density = cos(y);

/* (2) separate the undulating slab into fibers via cos(z) */

   density = density * cos(z);

/* (3) shape the boundary into a hard surface */

   density = density * density;
   density = bias(density, Bias);
   density = gain(density, Gain);
   return density;
```

We can then define a `woof` function by rotating 90 degrees in `z`, `x`, and flipping in `y`. The complete cloth function is then

```
    cloth(x, y, z) = warp(x, y, z) + warp(z, -y, x);
```



(a) (b) (c) (d)

FIGURE 12.7 Steps in the simulation of a sparsely fractal material.

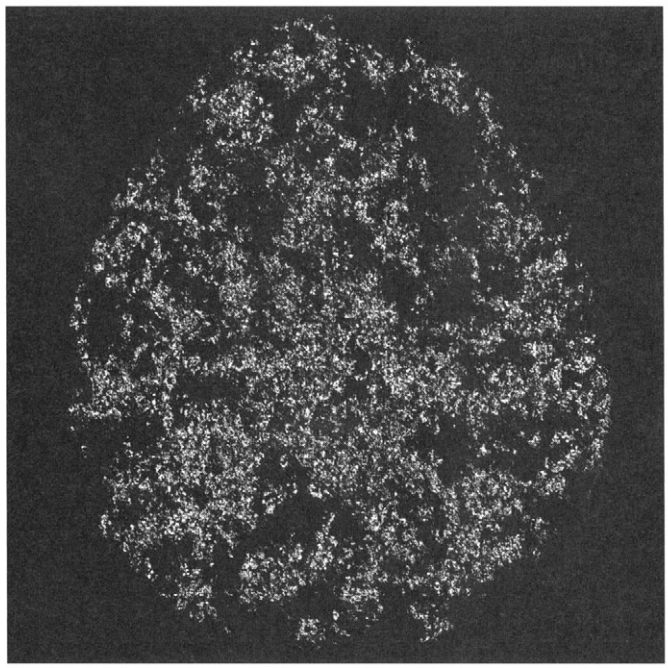


FIGURE 12.8 A shape having infinite surface and zero volume.

You can make the cloth wrinkle, fold, and so on, by transforming `x`, `y`, and `z` before applying the cloth function. You can also add high frequency *noise*() to `x`, `y`, `z` before applying `cloth()`, to simulate the appearance of roughly formed fibers. In the examples shown I have done both sorts of things.

In the cloth examples shown here, I "sharpen" the surface by applying the `bias()` and `gain()` functions. Figure 12.9(a) through 12.9(d) shows extreme close-ups of cloth with various bias and gain settings. Figure 12.9(a) has low bias and gain. In Figure 12.9(b) I increase gain, which "sharpens" the surface. In Figure 12.9(c) I increase bias, which expands the surface, in effect fattening the individual threads. In Figure 12.9(d) I increase both bias and gain. Figure 12.10 shows a high-resolution rendering of a low-bias, high-gain cloth, which gives a "thread" effect. Conversely, a high bias, low gain would give a "woolen" effect.

ARCHITEXTURE

Now let's take an architectural sketch and "grow" solid texture around it, ending up with hard textured surfaces. This is similar in spirit to Ned Greene's voxel automata

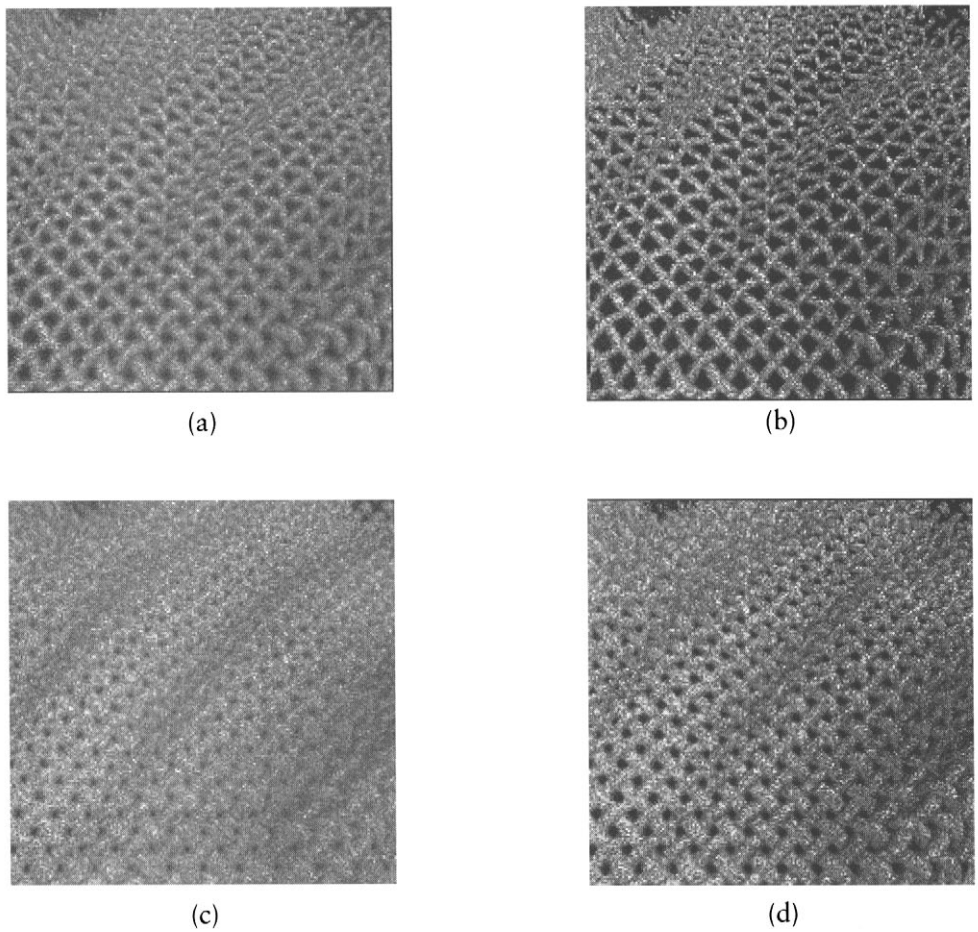


FIGURE 12.9 Extreme close-ups of cloth with various bias and gain settings.

algorithm. The difference is that whereas he literally "grows" a volume from a defining skeleton, one progressive voxel layer at a time, the hypertexture approach directly evaluates its result independently at each point in space.

I start with a skeleton of architectural elements. This can be supplied by freehand drawing or, alternatively, generated from a CAD program. Each architectural element is a "path" in space formed by consecutive points P_i.

Each path defines an influence region around it, which gives the architexture its shape component. This region is created by "blurring" the path. To do this I treat

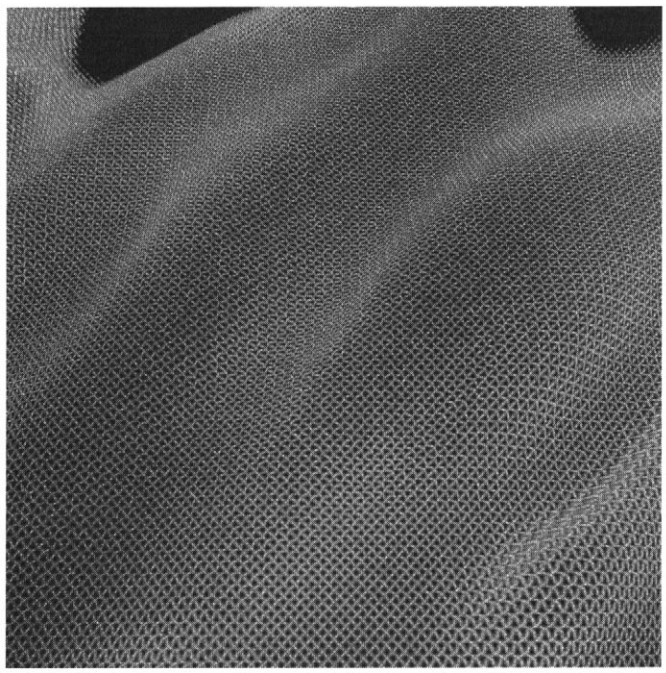


FIGURE 12.10 High-resolution rendering of a low-bias, high-gain cloth gives a "thread" effect.

each point along the path as the center of a low-density soft sphere of radius R. The shape density at a given point $\vec{x}$ is given by

$$\text{path\_shape}(\vec{x}) = \frac{1}{KR^2} \sum_i \max(0, R^2 - |P_i - \vec{x}|^2)$$

where the normalizing constant K is the distance between successive points on the path. For each volume sample, the cost per path point is a dot product and some adds, which is fairly expensive. To speed things up I maintain a bounding box around each path, which eliminates most paths from consideration for any given sample point.

I've only played so far with rocklike textures for architexture. The texture component of this is given by a simple noise-based fractal generator:

$$\text{rock\_texture}(\vec{x}) = \sum_{f=\log base\_freq}^{\log resolution} 2^{-f} noise(2^f \vec{x})$$

and I define the final density by

$$\text{sharpen}(\text{path\_shape}(\vec{x}) + \text{rock\_texture}(\vec{x}))$$

where I use the sharpening function to reduce the effective fuzzy region size about one volume sample. For a given image resolution and shape radius R, correct sharpen is done by

- scaling the density gradient about 0.5 by a factor of $1/R$ (adjusting also for variable image resolution)
- clipping the resulting density to between 0.0 and 1.0

The idea of the above is that the larger R becomes, the smaller will be the gradient of density within the fuzzy region, so the more sharpening is needed. The actual code I use to do this is

```
density = (density - 0.5) * (resolution / 600) / R + 0.5;
density = max(0.0, min(1.0, density));
```

I also decrease the shape's radius R with height (y-coordinate), which gives architectural elements a sense of being more massive in their lower, weight-supporting regions.

In Figures 12.11 and 12.12 I show a typical arch (which I suppose could archly be called archetypical architexture). This started out as a simple curved line tracing the inner skeleton, which was then processed as described above.

Figure 12.11 shows a sequence where three parameters are being varied. From left to right the width of the arch at the base increases. From top to bottom the thickness enhancement toward the base increases. These two parameters act in concert to add a "weight-supporting" character to architextural elements. Finally, the amplitude of the texture is increased linearly from the first image in the sequence to the last. Figure 12.12 shows a high-resolution version of the bottom-right image in the sequence.

The NYU Torch

In Figure 12.13, a number of hypertextures were used for various parts of the same object to build an impression of this well-known New York University icon. The various amounts of violet reflected down from the flame to the torch handle were just added manually, as a function of y.

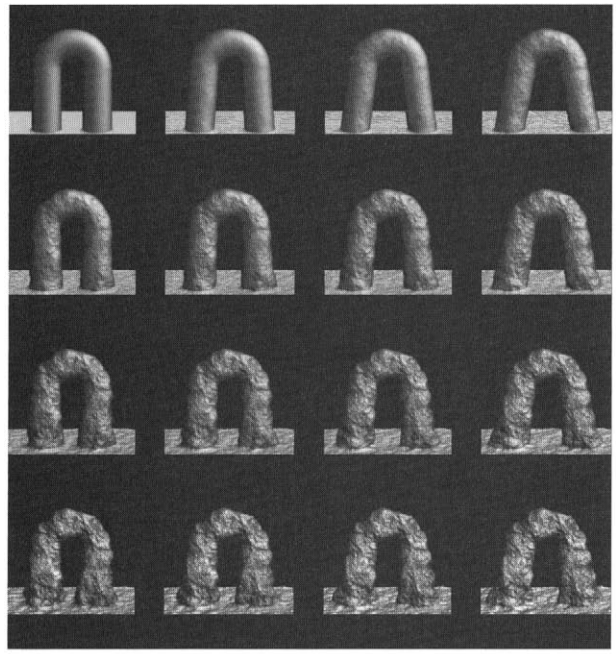


FIGURE 12.11 Arch with varying parameter values.

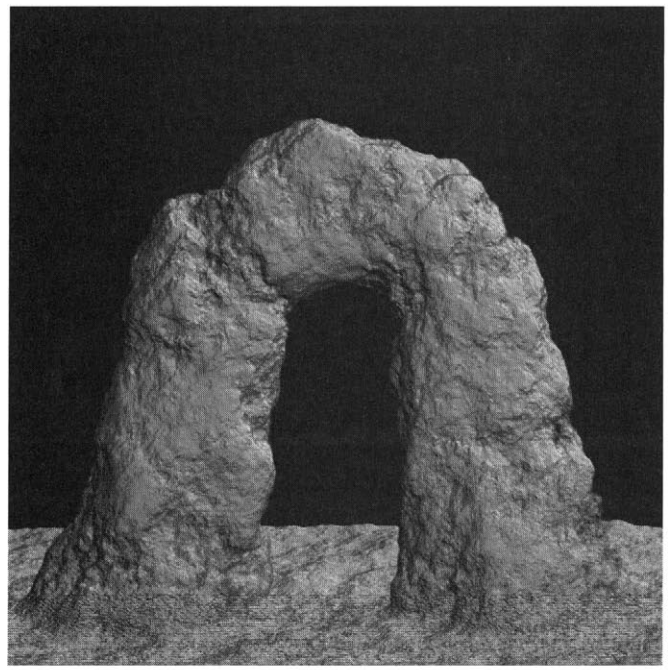


FIGURE 12.12 High-resolution version of the last image in the arch sequence.

FIGURE 12.13 New York University torch.

Smoke

In a recent experiment we tried to create the look of an animating smoke column, using as simple a hypertexture as possible. This work was done in collaboration with Ajay Rajkumar at NYU.

The basic approach was to create a smooth smoke "column" along the y-axis and then to perturb this column in x,z—increasing the perturbation at greater y

values. We added knobs for such things as column opacity and width. These "shaping" knobs can have a great effect on the final result. For example, Figure 12.14(a) and 12.14(b) vary only in their column width. Yet the difference in appearance between them is drastic.

Time Dependency

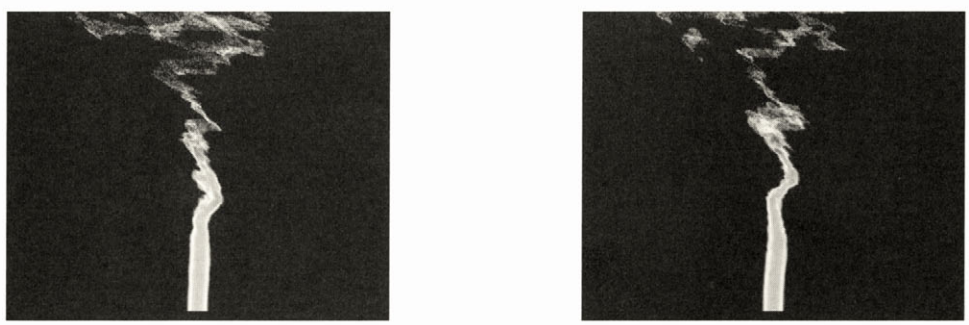
We make the smoke appear to "drift" in any direction over time by moving the domain of the turbulence in the *opposite* direction. In the following example, we do this domain shift in both x and y.

We move y linearly downward, to give the impression of a rising current. We move x to the left but increase the rate of movement at greater y values. This creates the impression that the smoke starts out moving vertically, but then drifts off to the right as it dissipates near the top.

The particular shape of the smoke can vary dramatically over time, yet the general *feel* of the smoke stays the same. Compare, for example, the two images in Figure 12.15(a) and 12.15(b), which are two frames from the same smoke animation.

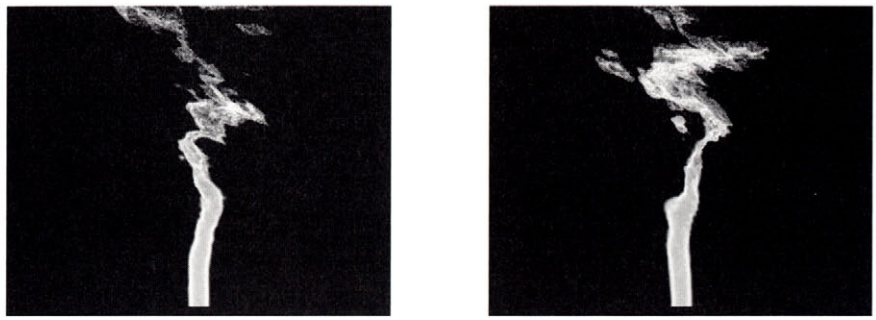
Smoke Rings

Figure 12.16 shows the formation over time of a smoke ring. Smoke rings will occur when the turbulence function distorts the space sufficiently so that the column appears to double over on itself horizontally. We need to let this happen only fairly high up on the column. If it happens too low, then the rings will appear to be forming somewhere off in space, not out of the column itself.



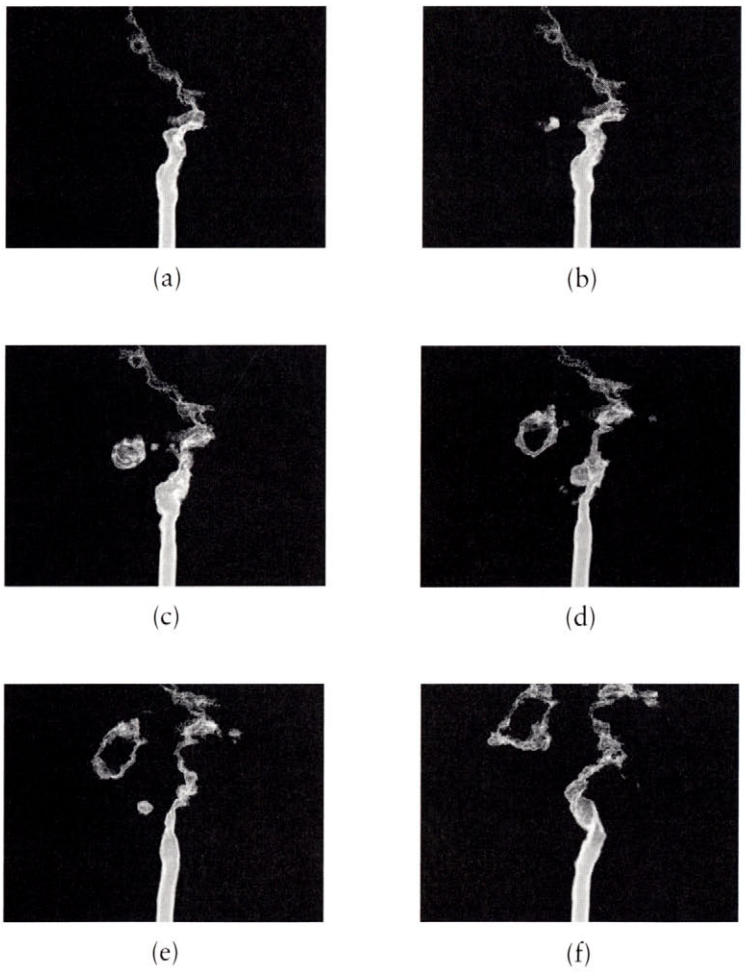
(a) (b)

FIGURE 12.14 Animating smoke column using simple hypertexture.



(a) (b)

FIGURE 12.15 Appearance of "drift" in smoke animation.



(a) (b)

(c) (d)

(e) (f)

FIGURE 12.16 Formation of a smoke ring.

For this reason, we employ two different gain curves. One controls turbulence amplitude near the column as a function of y, and the other controls turbulence amplitude far from the column as a function of y. The latter curve always lags behind the former, thereby preventing low-lying smoke rings.

Optimization

Since smoke is quite sparse within its sampled volume, it is a good candidate for optimization based on judicious presampling. Tests on the AT&T Pixel Machine with 64 DSP32 processors showed that it took about 8 hours to compute a single frame of a $640 \times 480 \times 640$ volume. This is rather impractical for animation. To speed this up, we precompute the image at a smaller resolution to find out where the smoke lives within the volume. We then do the final computation only within those parts of the volume.

More specifically, we do a preliminary raymarch at one-fourth the final x, y, z resolution. Note that this requires only 1/64 as many density evaluations as would a full computation. At each 4×4 pixel, we save the interval along z that bounds all nonzero densities. For many pixels, this will be a null interval. Then to be conservative, at each pixel we extend this interval to be its union with the intervals at all neighbor pixels.

We use this image of bounding z intervals to restrict the domain of the final raymarching. We have found about a 30-fold speedup using this method—each image now takes about 16 minutes to compute (including the time for the subsampled prepass). Smoke is optimal for this type of speedup for two reasons: (1) since it is sparse, the speedup is great, and (2) since density takes many sample widths to fall off to zero, tiny details are not inadvertently skipped over.

Here is pseudocode for the smoke density function. It's mostly just C code with some unimportant details and declarations left out.

```
smoke_density_function(x, y, z)
{
     /* k1, k2, etc. are the column shape knobs */

     /* rotate z randomly about y, based on noise function */
     /* this creates the impression of random "swirls" */

     t = noise(x,y,z);
     s = sin(t / 180. * PI);
     c = cos(t / 180. * PI);
     z = x * s + z * c;
```

```
    /* once the space is "swirled", create the column of smoke */

    /* 1) phase-shift x and z using turbulence; this varies with time */

    x += k1 * phase_shift(x, y, z);
    z += k2 * phase_shift(x, y, z + k3);

    /* 2) define column by distance from the y-axis */

    rx = (x * x + z * z) * k4;

    /* 3) if inside the column, make smoke */

    if (rx < =1.) {
        rx = bias(rx, k5); /* the basic column shape is */
        s = sin(PI * rx); /* a tube with hollow core */
        return s * s;
    }
    else
        return 0.;
}

phase_shift(x, y, z)
float x, y, z;
{
    /* c1, c2, etc. are the "texture" knobs */

    p[0] = c1 * x + bias(y + .5, .3) * TIME; /* vary with time */
    p[1] = c1 * y + TIME;
    p[2] = c1 * z + c2;
    g = gain(y + .5, c3); /* dropoff with y */

    /* these 3 lines remove smoke rings that are */
    /* too low in y to be physically plausible */

    r = max(0., 1. - (x * x + z * z) * c5);
    gl = gain(bias(y + .5, c4), c3);        /* smoke ring dropoff with y */
    g = g * LERP(gl, r, 1.);

    return g * (turbulence(p, 1., RES) + c6); /* c6 recenters the column */

}
```

TURBULENCE

The turbulence function, which you use to make marble, clouds, explosions, and so on, is just a simple fractal generating loop built on top of the *noise* function. It is not

a real turbulence model at all. The key trick is the use of the `fabs()` function, which makes the function have gradient discontinuity "fault lines" at all scales. This fools the eye into thinking it is seeing the results of turbulent flow. The `turbulence()` function gives the best results when used as a phase shift, as in the familiar marble trick:

```
sin(point + turbulence(point) * point.x);
```

Note the second argument in the following code, `lofreq`, which sets the lowest desired frequency component of the turbulence. The third argument, `hifreq`, is used by the function to ensure that the turbulence effect reaches down to the single pixel level, but no further. I usually set this argument equal to the image resolution.

```
float turbulence(point, lofreq, hifreq)
float point[3], freq, resolution;
{
    float noise3(), freq, t, p[3];

     p[0] = point[0] + 123.456;
     p[1] = point[1];
     p[2] = point[2];

     t = 0;
     for (freq = lofreq ; freq < hifreq ; freq *= 2.) {
         t += fabs(noise3(p)) / freq;

         p[0] *= 2.;
         p[1] *= 2.;
         p[2] *= 2.;
     }
     return t - 0.3; /* readjust so that mean returned value is 0.0 */
}
```

ANTIALIASED RENDERING OF PROCEDURAL TEXTURES

This section describes a way of antialiasing edges that result from conditionals in procedurally defined images, at a cost of one sample per pixel, wherever there are no other sources of high frequencies. The method proceeds by bifurcating the calculation over the image at conditional statements. Where a pixel straddles both true and false branches under its convolution area, both branches are followed and then linearly combined. A neighbor-comparing, antialiased, high-contrast filter is used to define regions of bifurcation. The method proceeds recursively for nested conditionals.

Background

If an image is described by an algorithmic procedure, then in principle there are cases where antialiasing can be done analytically, without supersampling, just by examining the procedure itself.

Generally speaking, there are two sources of high frequencies for procedurally generated images. *Edge events* are caused by conditional statements that do Boolean operations on continuous quantities. For example, there are infinitely high frequencies on the image of a circular disk formed by the following conditional statement when sampled over the unit square:

$$\text{if } (x - .5)^2 + (y - .5)^2 < .25 \text{ then } \textbf{white} \text{ else } \textbf{black}$$

The edges of discontinuity in this image can, in principle, be detected by analyzing the statement itself. Nonedge high-frequency events cannot be detected in this way. For example, to render the image represented by

$$\text{if } \sin(10^5 x) \text{ in}(10^5 y) > 0 \text{ then } \textbf{white} \text{ else } \textbf{black}$$

we need to resort to statistical oversampling or other numerical quadrature methods, since this expression has *inherently* high frequencies.

The Basic Idea

As in Perlin (1985), consider any procedure that takes a position (x, y) in an image as its argument and returns a color. Usually, we create an image from such a procedure by running the procedure at enough samples to create an antialiased image. Where the results produce high variance, we sample more finely and then apply a weighted sum of the results to convolve with a pixel reconstruction kernel.

One common source of this high local variance is the fact that at neighboring samples the procedure follows different paths at conditional statements. No matter how finely we oversample, we are faced with an infinitely thin edge on the image—a step function—between the **true** and **false** regions of this conditional.

Our method is to identify those edges on the image caused by conditional expressions. We do this by comparing neighboring sample values at each conditional expression in the computation. We use the results of this comparison to create an antialiased filter that represents the resulting step function as though it had been properly sampled.

To do this, we can view the procedure as a single-instruction-multiple-data (SIMD) parallel operation over all samples of the image. More precisely, we view the

computation as a reverse Polish stack machine. Items on the stack are entire images. For example, the "+" operator takes two images from the stack, adds them sample by sample, and puts the result back on the stack.

This calculation proceeds in lockstep for all samples. When a conditional is reached, we can evaluate both sides of the conditional and then just disregard the results from one of the branches.

Obviously, this approach is wasteful. Instead, we would like to go down only one path wherever possible. We arrange to do this as follows. We recast each conditional expression using a pseudofunction *ifpos*, so that *expr ifpos* evaluates to 1 when *expr* > 0 and 0 otherwise. Using the *ifpos* operator we can recast any function containing conditionals into an expression

$$expr\ ifpos\ [ToDoIfFalse,\ ToDoIfTrue]\ LERP$$

where t [a, b] *LERP* is a linear interpolation operation defined as follows: when $t \leq 0$ or $t \geq 1$, *LERP* returns a or b, respectively. When $0 < t < 1$, *LERP* returns $a + t(b - a)$. For example:

$$abs(x) = \text{if } x < 0 \text{ then } -x \text{ else } x$$

can be expressed as

$$x\ ifpos\ [-x\ ,\ x\ ]LERP$$

Over the convolution kernel of any pixel, *ifpos* will return **true** for some samples and **false** for others, creating a step function over the image. We actually return a floating-point value between 0.0 and 1.0 to express the convolution of this step function with each sample's reconstruction kernel. Now the problem of properly sampling edges that have been formed from conditions is completely contained within one pseudofunction.

This is done as follows. When execution of our SIMD stack machine gets to an *ifpos* function, it looks at the image on top of the stack and runs a high-contrast filter h on this image, which examines the neighbors of each sample. If the sample and its neighbors are all positive or all negative, then h returns 1 or 0, respectively. Otherwise, h approximates the gradient near the sample by fitting a plane to the values at neighboring samples. Where there is a large variance from this plane, then supersampling must be done—the procedure must be run at more samples.

But where there is a low variance, then just from the available samples, h can produce a linear approximation to the argument of *ifpos* in the neighborhood of

the sample. It uses this to construct a step function, which it then convolves with the sample's reconstruction kernel. As we shall see, this is a fairly inexpensive procedure.

After the *h* function is run, the samples of the image on the stack will fall into one of three subsets:

$$value \equiv 0$$
$$0 < value < 1$$
$$value \equiv 1$$

We label these subsets *F*, *M*, and *T* (for "false," "midway," and "true"), respectively.

Once *h* is run, we gather up all samples in (*F* union *M*) and run the *ToDoIfFalse* statements on these. Then we gather up all samples in (*T* union *M*) and run the *ToDoIfTrue* statements on these. Finally, we do a linear interpolation between the two branches over the samples in *M* and place the reconstructed image on the stack.

Because each conditional splits the image into subsets, some of a sample's neighbors may be undefined inside a conditional. This means that within nested conditionals, *h* may not be faced with a full complement of neighbors. This happens near samples of the image where several conditional edges intersect. As long as there is at least one horizontal and one vertical neighbor, *h* can re-create the gradient it needs. When there are no longer enough neighbors, *h* flags a "high-variance" condition, and supersampling is then invoked.

More Detailed Description

In this section we describe the flow of control of the algorithm. We maintain a stack of lists of *current_samples*. We start with *current_samples* = ALL samples. At any given moment during execution, samples are examined only if they are in the current list.

(I) When the *ifpos* token is encountered:
Evaluate high-contrast filter for all samples in *current_samples*.
Use the result to make the three sublists: *F*, *M*, *T*.
Push these sublists onto an *FMT* stack.
Push a new *current_samples* list (*M* union *T*).

(II) Continue to evaluate normally, over samples in *current_samples*.
If the *ifpos* code is encountered, recurse to (I).

(III) When the beginning of the *ToDoIfFalse* code is encountered:
Pop the *current_samples* stack.
Push a new *current_samples* list (*F* union *M*).

Evaluate normally until matching *LERP* token is encountered.
If the *ifpos* token is encountered first, recurse to (I).

(IV) When the *LERP* token is encountered:
Pop the *current_samples* stack.
For all samples in *current_samples:*
stacktop[1] := LERP(stacktop[1], stacktop[−1], stacktop[0])
Pop the *FMT* stack.

The High-Contrast Filter

Given a large enough subset of the eight neighbors around a sample, we can use the value at the sample and its neighbors to construct an approximation to the integral under the sample's reconstruction kernel in the area where *ifpos* evaluates to 1.

Let $f(x, y)$ denote the value of the expression at any small offset (x, y) from the sample. First we do a least-squares fit of the differences in value between the sample and its neighbors to approximate the x and y partials. If the variance is high for either of these two computations, then we give up and resort to supersampling the procedure in the neighborhood of the sample.

Otherwise, we can use the linear approximation function $ax + by + c = 0$ near the sample, where $c = f(0,0)$, and a and b approximate the x and y partials of f, respectively.

We want to find out the integral under the sample's reconstruction kernel of the step function

$$\text{if } f(x, y) > 0 \text{ then 1 else 0}$$

We assume a circularly symmetric reconstruction kernel.

The problem at this stage is illustrated in Figure 12.17. Figure 12.17(a) shows the sample reconstruction kernel, Figure 12.17(b) shows the intersection of this kernel with a linear approximation of the function near the sample, Figure 12.17(c) shows the step function induced by this linear function, and Figure 12.17(d) shows the region under the kernel where this step function is positive.

Since we assume the kernel is circularly symmetric, finding the convolution reduces to a one-dimensional problem. We need only compute the perpendicular distance from the sample to the step and then use a spline fit to approximate the integral under the kernel in the positive region of the step function.

To do this, we compute the magnitude of the linear gradient

$$|f'| = \sqrt{(a^2 + b^2)}$$

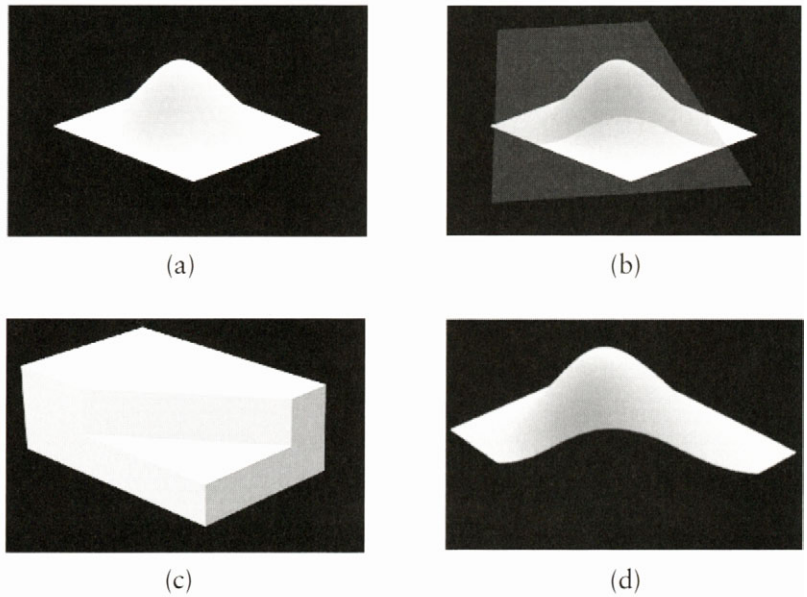


FIGURE 12.17 Reconstruction kernel.

Then we calculate the distance from the sample to the line of the image where this function equals zero by

$$d = f(x,y) / |f'|$$

Finally, we approximate the integral of the reconstruction kernel by a cubic spline:

$$\text{if } t < -.5 \text{ then } 0 \text{ else if } t > .5 \text{ then } 1 \text{ else } 3t + .5)^2 - 2(t + .5)^3$$

Examples

Figure 12.18(a) through 12.18(f) shows a succession of steps in the creation of a simple procedural image that requires two levels of nested conditionals. Each image is on the top of the stack during a particular snapshot of execution.

The image is formed by doing a conditional evaluation of the circular disk (Figure 12.18(a)):

$$(x - .5)^2 + (y - .5)^2$$

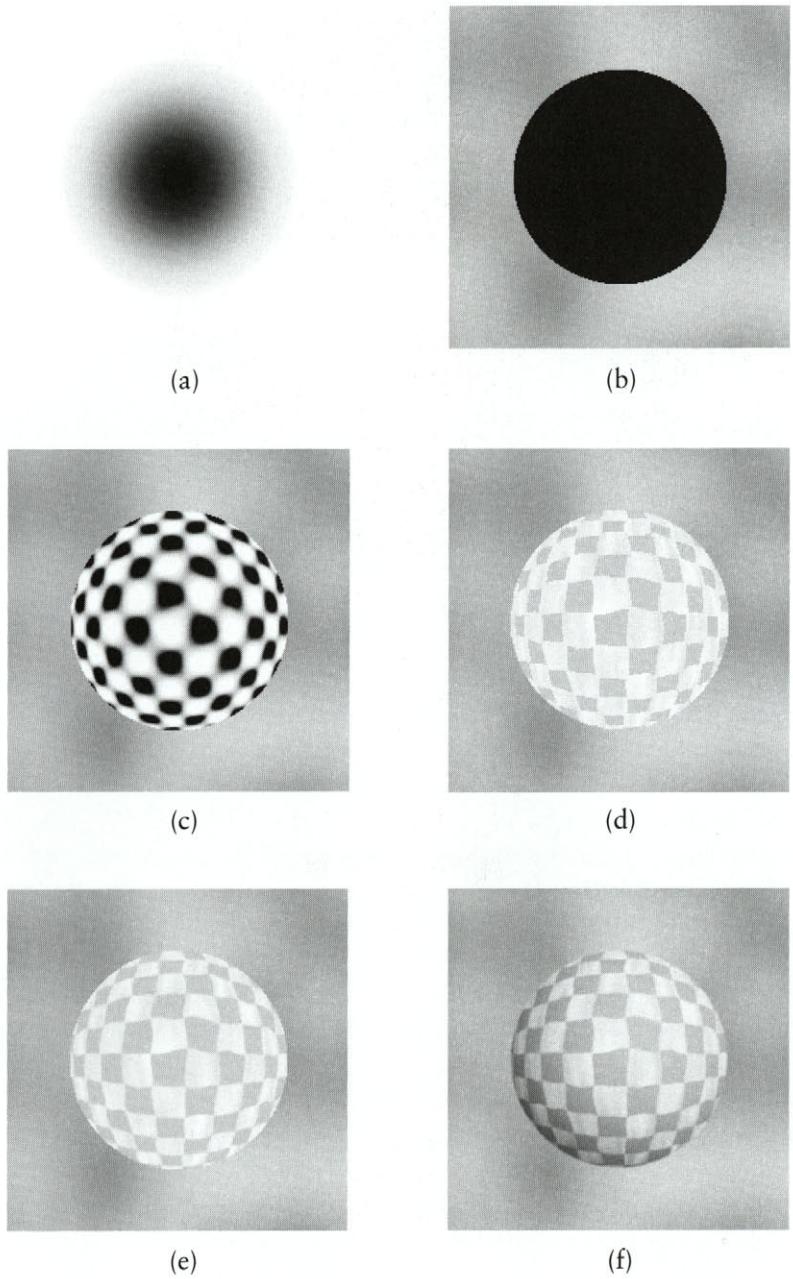


FIGURE 12.18 Creation of a simple procedural image that requires two levels of nested conditionals.

Then $.3^2$ is subtracted, a conditional is done, execution bifurcates, and *noise* (Perlin 1985) is applied to the outside of the disk (Figure 12.18(b)). Then $\sin(10x + noise(x, y)/10)\sin(10y + noise(x, y)/10)$ is evaluated inside the disk (Figure 12.18(c)), and another conditional is done. A constant is applied to the negative areas, and $.5 + noise(3x, y)$ is applied to the positive areas (Figure 12.18(d)). Finally, Figure 12.18(e) and 12.18(f) show the successive reconstruction of the two nested levels of conditionals.

To Sum Up

The previous method recasts image synthesis as a set of recursively bifurcating SIMD processes over image samples. In this way we were able to do true procedural antialiasing of edges.

This approach differs from previous approaches in that it finds and resolves sampling problems by using information contained in the image synthesis algorithm itself, usually without resorting to supersampling. Whether such methods can be applied to the general image synthesis problem is an open question.

SURFLETS

Instead of regenerating hypertexture procedurally all the time, it is sometimes useful to "cache" it in some format for display. For this purpose we want rapid and accurate display, as well as a good ability to do light propagation—including self-shadowing and diffuse penumbra.

This section describes a useful intermediate representation for procedurally generated volumes. The approach is based on a sparse wavelet representation and is particularly suitable for the sorts of free-form surfaces generated by hypertextural algorithms.

Other sorts of complex volumetric data can also be conveniently represented using this approach, including medical data (CT and MR) and meteorological, geological, or molecular surface data.

A *surflet* is a flavor of wavelet that can be used to construct free-form surfaces. Surfaces obtained by evaluating scalar volumes or procedural volumetric models such as hypertexture can be stored in the intermediate form of surflets. This allows us when rendering such surfaces to (1) do self-shadowing, including penumbras, from multiple light sources, and (2) have the option at rendering time to either (a) further refine the surface locally from the volume data or (b) do a bump texture approximation or any mixture of (a) and (b). Thus issues of lighting placement and level-of-details differences can be resolved at a postprocessing stage, after the bulk of

the computation is completed. This is work I did mainly with Benjamin Zhu and is based in part on earlier work that I did together with Xue Dong Yang.

First we will define the surflet model and show a particular implementation. Then we will discuss how to find visible surfaces. We will show how, by using surflets, we can decide locally at rendering time whether true volumetric surface refinement or shading approximation should be performed. Finally, we will introduce a multiresolution, hierarchical modeling of surflets and describe a way to do self-shadowing with penumbra.

Introduction to Surflets

Volume visualization studies the manipulation and display of volume data. Depending on the intermediate representation between raw data and final display, volume visualization techniques can be classified as surface-based techniques (or surface modeling) (Lorensen and Cline 1987, 1990; Wyvill, McPheeters, and Wyvill 1986) and volume rendering (Drebin, Carpenter, and Hanrahan 1988; Hanrahan 1990; Kajiya and Von Herzen 1984; Levoy 1988, 1990a, 1990b, 1990c; Sabella 1988; Westover 1990). Methods in the first category approximate surfaces of interest (determined by thresholds) by geometric primitives and then display these primitives, whereas volume rendering directly operates on the volume data and renders the scenes.

Surface modeling is the main topic of this discussion. Since surfaces offer a better understanding of the external structure of volumes, accurately reconstructing surfaces from volumes and subsequently rendering the surfaces efficiently is very important in practice. Surface modeling techniques differ from one another in that different primitives can be used to approximate the surfaces of interest. Keppel (1975) and Fuchs (Fuchs, Kedem, and Uselton 1977) construct contours on each 2D slice and connect contours on subsequent slices with triangles. The cuberille model uses parallel planes to create surfaces (Chen et al. 1985; Gordon and Reynolds 1985; Hoehne et al. 1990). The marching-cube algorithm uses voxel-sized triangles to approximate surfaces (Cline, Lorensen, and Eudke 1988; Lorensen and Cline 1987, 1990; Wyvill, McPheeters, and Wyvill 1986). Bicubic patches can also be used to reconstruct surfaces (Gallagher and Nagtegaal 1989).

All of these techniques share a common characteristic: the geometric primitives involved in surface reconstruction are in an explicit form. This leads to an interesting question: Can we define the surfaces of interest in an implicit form? If so, what are the advantages of using the implicit representation?

In the following sections, a "surflet" model is introduced to define free-form surfaces. Those sections will describe methods to find visible surfaces, perform selective

surface refinement, and do self-shadowing with penumbra. A multiresolution, hierarchical modeling of isosurfaces is sketched. Experimental results are shown to demonstrate the benefits of adopting this new model.

Surflets as Wavelets

As discussed earlier in the section on *noise*, a wavelet (Mallat 1989a, 1989b) is a function that is finite in extent and integrates to zero. Decomposing signals or images into wavelets and then doing analysis on the wavelet decomposition is a powerful tool in signal and image recognition.

We use a specific kind of continuous wavelet, which we call surflets, to approximate surfaces. We use a summation of surflets that together define a function over R^3 so that the desired surface is closely approximated by the locus of points where (i) the summation function is zero but where (ii) the function's gradient magnitude is nonzero. The general intuition here is that each surflet is used as a free-form spline that approximates a little piece of localized surface. All of these surflets sum together to form isosurfaces of interest.

We define each surflet as follows: let $\vec{p} = [x, y, z]$ be a sampled location in space and let r be its sampling radius. Then if the sample at $\vec{p}$ has a value of d and a gradient vector of $\vec{n}$, we define a wavelet approximation to the sample at $\vec{x} = [x, y, z]$ near $\vec{p}$ by

$$\prod drop\left(\frac{x_i - p_i}{r}\right) \times \sum n_i(x_i - p_i)$$

where

- $\vec{x}$ is any point in R^3
- $\vec{p}$ is the wavelet center
- $\vec{n}$ is the wavelet gradient
- r is the wavelet radius
- i varies over the three coordinates
- $drop(t)$ is defined as for the *noise* function: $2t^3 - 3t^2$

At points with distance greater than r from $\vec{p}$ we assign a value of zero.

$[\vec{p}, d, \vec{n}]$ defines a surflet. Since each sample at $\vec{p}$ cannot affect any sampled location farther than r away from its center, the wavelet contribution is localized.

It should be pointed out that other continuous functions might also work well, as long as each surflet has a limited extent in terms of its wavelet contribution, and this contribution drops from 1.0 down to 0.0 monotonically. The cubic drop-off function is chosen due to its simplicity.

In practice, we define the surflets on a rectangular grid. When surflets are refined (see ahead), the size of the sampling grid is successively halved. We generate surflets from a functional sampling with the following steps:

- Subtract the desired isovalue from all samples.
- Find samples that have any six-connected neighbor with density of opposite sign.
- Approximate surflet normal $\vec{n}$ by density difference between neighbors.
- Approximate surflet normal $\vec{n}$.
- Use $\vec{n}$ to locate center $\vec{p}$ near the sample.

We use the normal to locate the surflet center as follows:

- Approximate surflet normal $\vec{n}$ by taking density difference between neighbor samples along each coordinate axis.
- Use $\vec{n}$ to locate surflet center $\vec{p}$:

$$\vec{p} = \vec{x}_{\text{sample}} - d_{\text{sample}}\vec{n}$$

Note that we end up with something very closely related to the *noise* function (Perlin 1985). In fact, the *noise* function can be viewed as the surflet decomposition of a random density function sampled at a rectangular grid.

Finding Visible Surfaces

Because surflets are well-defined analytic functions, it is possible to find the intersection of a ray with the zero surface of a summation-of-surflets function.

Since the contribution from each surflet has a limited extent, for each point $\vec{x}$ in R^3 only a finite number of surflets will have a nonzero wavelet contribution to $\vec{x}$. Two alternatives are possible for finding a ray-surface intersection. A numerical method such as binary division or Newton iteration can guarantee convergence to the true intersection, given initial guesses close to the actual solution. However, since all these numerical methods involve iterative root finding, they can be quite slow. We instead use a faster approximation method.

As in cone tracing (Amanatides 1984), we trace a fat ray (a cone) into the scene. We represent each surflet by a sphere whose radius is the surflet's r times a constant factor and whose center is $\vec{p}$. We shrink the cone ray into a thin line and grow all the spheres by the cross-sectional width of the cone at the surflet. This is equivalent to the computation in Amanatides (1984). Since perspective projection is used, the spheres that approximate the surflets become variously sized. Each sphere is given a weight according to how far along its radius it intersects the ray as well as the magnitude of its gradient. Spheres that mutually intersect each other within the ray form a local bit of surface. We use a scheme similar to the fragment merging in Carpenter's A-buffer algorithm (Carpenter 1984) and use the weighted average of the surflet normals as the surface fragment normal. All such surface fragments are evaluated from front to back along the ray as opacity accumulates.

Shading is done as in Duff (1985). We stop raymarching when the opacity reaches totality. Because we render one surflet at a time (as opposed to one ray at a time), our surface approximation method is view dependent, in contrast to the numerical approach, which is view independent.

Visible surfaces can be rendered efficiently by using a depth-sorting algorithm similar to the Z-buffer algorithm. More efficiency can be derived by using the selective surface refinement described in the following section and the hierarchical surflet modeling described in the section "Constructing a Surflet Hierarchy."

Selective Surface Refinement

A sampled location is defined as a singularity if, along any coordinate axis, it is inside the surface while its two neighbors are outside the surface, or it is outside the surface while its two neighbors are inside the surface.

We generally need to refine a surflet only when any one of the following conditions is true:

- Due to perspective, the surflet is large with respect to a pixel on the viewing plane.
- The surflet normal is nearly perpendicular to the line of sight, thus presenting a silhouette edge.
- The surflet is a singularity.

However, if only the first condition is satisfied, then a surflet, no matter how large, can be visually approximated by normal perturbation alone.

By using selective surface refinement, we can greatly reduce the rendering time. Since surface refinement adapts to the complexity of a scene, different parts of the scene can be rendered at different resolutions. This is very attractive in practice. We have observed that the number of surflets involved in rendering decreases by more than 50% when we start at a low resolution and perform selective surface refinement at the next higher resolution than when we start directly at the next resolution level.

A Surflet Generator

Surflets can be generated from either sampled scalar data or procedural volumetric functions such as hypertexture. In the first case, we read in digitized samples from a regular grid; in the latter case, we evaluate samples on a regular grid. One orthogonal 2D slice is processed at a time. We always keep three slices, which helps us detect surflets as well as singularities.

After each slice has been processed, we find those samples that have any neighbors with density value on the opposite side of the isosurface density. At a candidate sample located at point $\vec{p}$, we approximate the normal gradient $\vec{n}$ by the density difference d between the samples and its neighboring samples along each coordinate axis. Forward difference, central difference, or a mixture of both (Hoehne et al. 1990) can be applied to get gradients. If the distance from the sample to the surface is less than 0.5, then $[\vec{p}, d, \vec{n}]$ defines a surflet. This 0.5 bias is determined empirically to compromise between two constraints: (1) If the distance is too small, we do not have enough surflets to interlock with each other, and consequently we might get undersampling. (2) If the distance is too big, then we will have too many surflets. This will create a huge intermediate structure to store surflets, and the rendering time will be too high.

For those samples that are singularities, each one is treated as eight different samples in eight octants; their gradient vectors and distances to the isosurface are evaluated. Therefore, a maximum number of eight surflets can be generated. Each singular surflet will have an appropriate flag tagged in its data structure, indicating which octant it contributes to.

Constructing a Surflet Hierarchy

Up to now we have described surflets limited to a single resolution. However, it would be more desirable to create surflets iteratively from lower resolutions to higher resolutions. A hierarchical surflet model not only can provide the freedom to

render surfaces at arbitrary levels of detail but also can avoid unnecessary details that would be washed out due to shadowing and so on.

We construct a surflet hierarchy as follows: Since only the isosurfaces are interesting to us, a candidate set of potential surflets is created at each resolution as in the previous section. Each candidate set is generally very small as compared to the number of samples in the volume. All surflets at the lowest resolution level are detected and collected. We compute the image at this resolution based on the wavelet approximation formula. Then we proceed to the next higher level and subtract the image at this level from the image at the previous level. Because we have precomputed the surflet candidate set at this level, the computation only involves those potential surflets. Those samples less than $r/2$ away from the isosurface are classified as surflets, where r is the sampling radius at this resolution. We repeat the same procedure and subtract images at lower resolutions from higher resolutions. When the sampling resolution equals the size of the sampling grid, a surflet hierarchy is created.

Self-Shadowing with Penumbra

Let us assume for simplicity that there is a single light source in the scene. We can then handle multiple light sources as the summation of the shading contribution from each light source.

We want to render surfaces with self-shadowing and penumbra. It is theoretically impossible to use a point source to create true penumbra, although some filtering techniques might create quite vivid penumbra effects (Reeves, Salesin, and Cook 1987). We instead use a light source of a certain shape, such as a spherical light source (or a rectangular light source). The portion of light source visible from any surflet is computed and used to determine the brightness at a surflet. An approximation of this portion is sketched here.

Each surflet in the scene is approximated by a sphere whose radius r is the sampling radius and whose center is given by $\vec{p}$. For each visible surflet, we trace a fat shadow ray from the surflet center to the light source. The spread angle of the fat ray is approximated by R/D, where R is the radius of the spherical light source and D is the distance from the surflet center to the center of the light source. We perform cone tracing to determine shadowing and penumbra in the same way that we found visible surfaces. We shrink the cone ray into a thin line and grow the spheres accordingly. The portion of the shadow ray that any surflet blocks is determined by how far along its radius the surflet intersects the ray. To make the approximation more accurate, spheres that mutually intersect each other within the ray are split into small,

disjoint fragments. Portions of the shadow ray blocked by all these small segments are computed and summed to find the proportion of light blocked by this piece of surface.

We compose the portion of the shadow ray blocked by isosurfaces by a method similar to the image composition scheme in Duff (1985). The shadow ray marches through the scene until (1) the blocked portion reaches totality or (2) the shadow ray reaches the light source. In the first case, the surflet that initiates the shadow ray is in full umbra. In the latter case, we subtract this blocked portion from 1.0 to get the surflet illumination. Clearly, the surflet is in penumbra if its illumination level is less than 1.0, and the surflet is not in shadow if its illumination equals 1.0.

To speed up the rendering, we can render surflets in shadow at lower resolutions. For example, we can do shadow detection at low resolutions and do selective surface refinement only for those visible surflets not in shadow. A major advantage of this scheme is that important surface details are not left out, while unnecessary details hidden by the shadow are not given much attention. This can produce softer shadows, as well as increase rendering speed.

Discussion

The surflet model has a number of advantages over other surface-based techniques. First, it has an implicit form. Although we approximate surflets with spheres in our implementation, it does not mean that this is the only choice. On the contrary, it is not clear to us whether this is the best way. For example, with special hardware, numerical root finding can be more accurate and more promising. We have also started to experiment with ellipsoid-like primitives to approximate surflets.

Second, the surflet model provides a convenient way to do hierarchical modeling of surfaces and selective surface refinement due to its implicit form. This feature cannot be found in many existing surface modeling methods. Adaptive sampling gives us the power to avoid unnecessary details while preserving important surface subtleties.

Third, the representation has a compact structure. Our experiments indicated that using surflets takes only 25% to 50% of the storage of marching cubes at the same resolution.

Fourth, isosurfaces can be rendered in parallel with surflets. Since for any point on the isosurfaces there are only a limited number of surflets determining the zero crossing, surflets are amenable to either a parallel implementation or an implementation with distributed computing.

Fifth, surflets can be integrated with wavelets in a straightforward manner to yield a combined model of surface modeling and volume rendering. If we delay thresholding until the rendering stage, we can generate wavelets by subtracting the low-resolution signals from the high-resolution signals with the same kind of hierarchical modeling as in the section "Constructing a Surflet Hierarchy." Depending on our need, we can do either volume rendering or thresholding followed by surface rendering at rendering time. This integrated approach is attractive, since it reduces the difference between rendering surfaces and rendering volumes. However, it differs from conventional volume rendering (Drebin, Carpenter, and Hanrahan 1988; Levoy 1988, 1990c) in that an intermediate data structure, as well as hierarchical modeling, is introduced to speed up the process. Thresholding can still be used at rendering time to distinguish the rendering from volume rendering. Moreover, it is possible to have volume details and surface details in the same scene.

Conclusion

Surflets are a free-form modeling of isosurfaces. This model is attractive in that it provides a convenient way to do shadowing, selective surface refinement, and hierarchical modeling. Moreover, it requires much less storage than other volume-to-surface methods and allows considerable freedom for a particular implementation. Finally, it encourages an integrated approach for surface modeling and volume rendering.

This surflet model is still quite empirical. Although it is quite intuitive and supported by image synthesis theory (Grossman and Morlet 1984; Mallat 1989a, 1989b), in many places we have had to tune the parameters to make the rendered images more realistic.

There is a lot of promise in integrating surface modeling with volume rendering. This kind of hybrid is very promising for hierarchical modeling of surfaces, since hierarchical modeling of volumes is more general than that of surfaces.

FLOW NOISE

In recent work with Fabrice Neyret (Perlin and Neyret 2001) we have been modifying the *noise* function so that it can give a suggestion of swirling and flowing time-varying textures.

Flow textures that use noise can look great, but they often don't "flow" right, because they lack the swirling and advection of real flow. We extend *noise* so that

shaders that use it can be animated over time to produce flow textures with a "swirling" quality. We also show how to visually approximate advected flow within shaders.

Rotating Gradients

Remember that we can think of the *noise* function as a sum of overlapping pseudo-random wavelets. Each wavelet, centered at a different integer lattice point (i, j, k), consists of a product of a weight kernel K and a linear function $(a, b, c)_{i,j,k}$. K smoothly drops off away from (i, j, k), reaching 0 in both value and gradient at unit distance. Each $(a, b, c)_{i,j,k} = a(x - i) + b(y - j) + c(z - j)$, which has a value of 0 at (i, j, k). The result of summing all these overlapping wavelets, *noise* has a characteristic random yet smooth appearance.

Our modification is to rotate all the linear vectors $(a, b, c)_{i,j,k}$ over time, which causes each wavelet to rotate in place. This process is illustrated in Figure 12.19.

Because all the (a, b, c) vectors were uncorrelated before the rotation, they will remain uncorrelated after the rotation, so at every moment the result will look like noise. Figure 12.20 illustrates this difference.

Yet over time, the result will impart a "swirling" quality to flow. When multiple scales of noise are summed together, we make the rotation proportional to spatial frequency (finer noise is rotated faster), which visually models real flow.

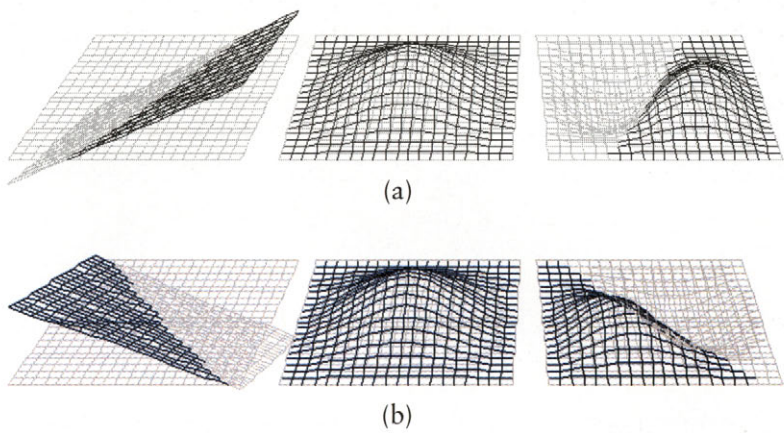


FIGURE 12.19 (a) Gradient * kernel ⇒ wavelet; (b) rotated gradiant * kernel ⇒ rotated wavelet.

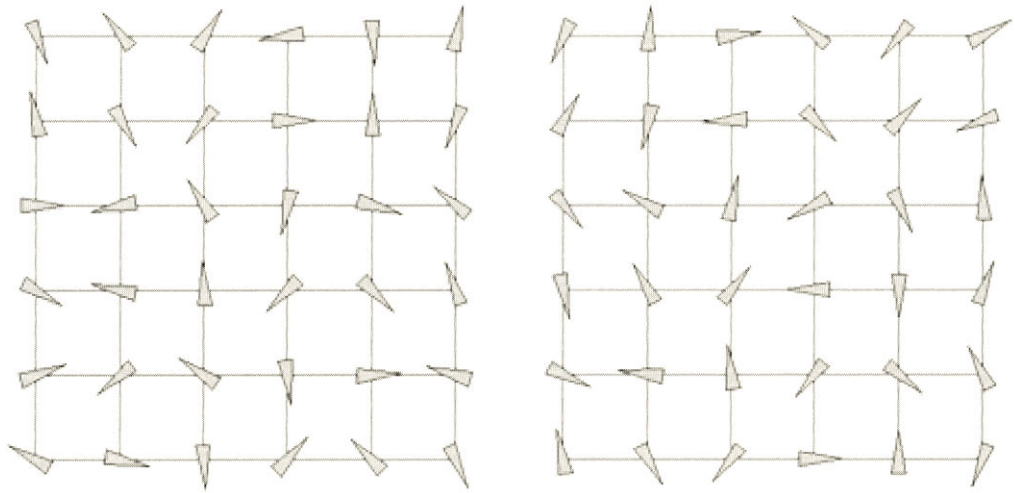


FIGURE 12.20 (a) Before rotation; (b) after rotation.

Pseudoadvection

Beyond swirling, fluids also contain advection of small features by larger ones, such as ripples on waves. This effect tends to stretch persistent features (e.g., foam), but not newly created ones (e.g., billowing), to varying degrees, according to their rate of regeneration, or *structure memory M*.

Traditional Perlin turbulence is an independent sum of scaled noise, where the scaled noise is

$$b_i(x) = noise(2^i x)/2^i$$

and the turbulence is

$$t_N(x) = \sum_{i=0}^{N} b_i(x)$$

This can define a displacement texture

$$color(x) = C(x + It_N(x))$$

where C is a color table and I controls amplitude. Our pseudoadvection displaces features at scale $i + 1$ and location x_0 in the noise domain to $x_1 = x_0 + k\, t_i(x_0)$, where k is the amplitude of the displacement (see below). For small displacements, this can be approximated by $x_1 - k\, t_i(x_1)$, so displacement k is proportional to an amplitude I specified by the user. We can scale this displacement by a "structure memory" factor M. We can simulate passive structures, which are totally advected, when $M = 1$, or very active structures, which are instantaneously generated and therefore unstretched, when $M = 0$. Our advected turbulence function is defined by modifying the scaled noise to

$$b_i(x) = b(2^i\,(x - IM\, t_{i-1}(x)))/2^i$$

and using this to construct the sum

$$t_N(x) = \sum_{i=0}^{N} b_i(x)$$

Results

Many flow textures can be created. A few are shown in Figures 12.21–12.24.

PROCEDURAL SHAPE SYNTHESIS

Recently, I've been working with Luiz Velho and others on multiresolution shape blending (Velho et al. 2001). In this approach, you conceptualize surface shapes as textures and then use multiresolution techniques to smoothly blend those shapes together, just as you might blend one texture into another. For example, just as Peter Burt (1983) originally showed that you can smoothly blend two images together by separately blending each of their multiscale components, you can do similar things with textural surface deformations. Two very interesting images we created using these techniques are shown in Figures 12.25 and 12.26.

TEXTURAL LIMB ANIMATION

In this section we borrow notions from procedural texture synthesis and use them to control the affect of humanlike figures. Stochastic *noise* is applied to create time-varying parameters with well-controlled statistics. We then linearly blend these parameters and feed the results into a motion system. Potential uses of this technique include role-playing games, simulated conferences, "clip animation," and simulated dance.

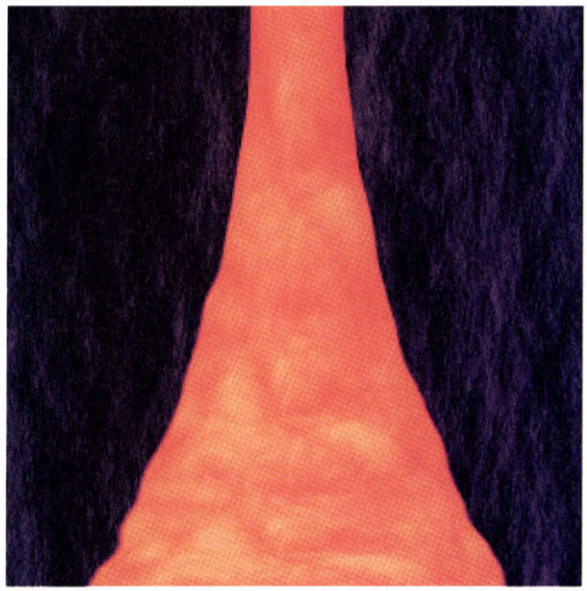


FIGURE 12.21 Lava flow.

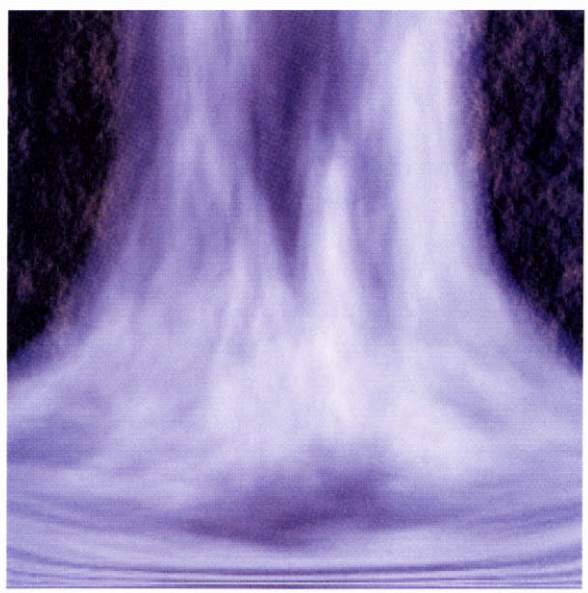


FIGURE 12.22 Waterfall.

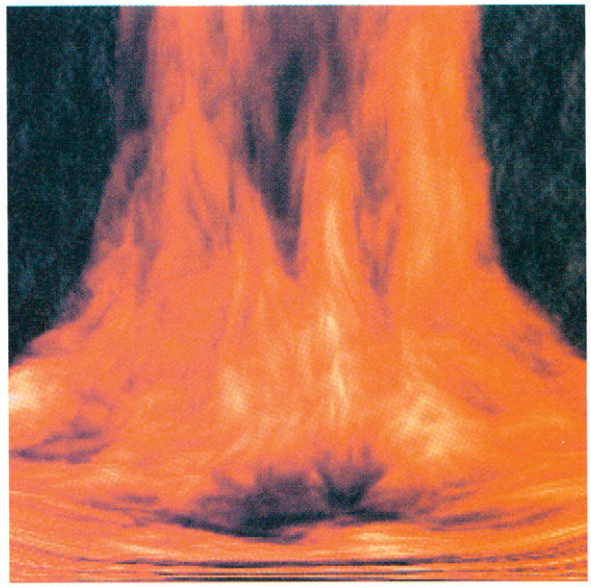


FIGURE 12.23 Waterfall with lava flow texture.

FIGURE 12.24 Swirling clouds using flow noise.

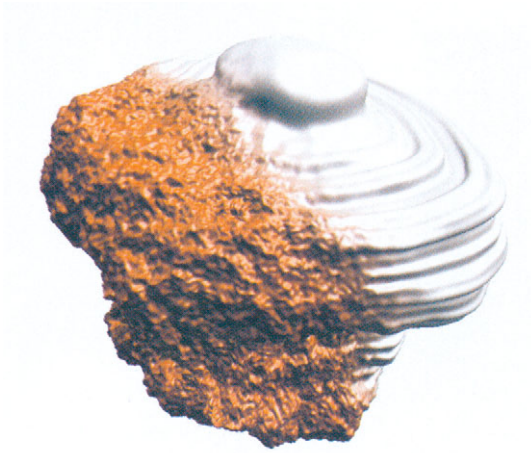


FIGURE 12.25 Rock blending into a screw shape.

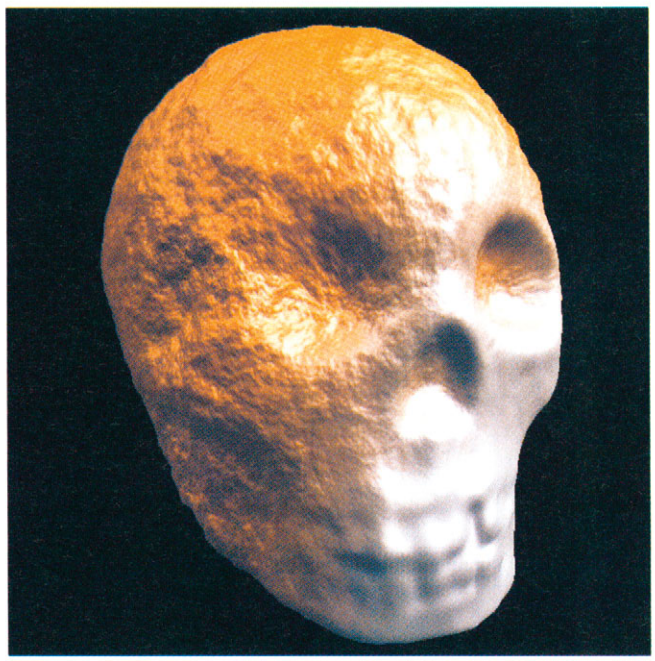


FIGURE 12.26 Corroded skull blending into a fresh one.

Introduction to Textural Limb Motion

In simulated environments we often want synthetic cooperating characters to respond to each other and to a changing environment in real time. Because simulated worlds are often used primarily as a means of communication (in contrast to, say, robotics simulation), we are often not so concerned with specific task completion as with conveying emotional messages through motion or gesture.

This situation comes up in role-playing games, where game players want their "Avatars" (Stephenson 1992) to show a particular affective response to a person or event in the immediate environment.

Similarly, emotive gesturing would be useful for monitoring a shared electronic "talk" conference. You could model the participants as physical gesturing characters around a conference table. At a glance you could tell who is talking to whom, who is entering and leaving the discussion, and who is paying attention to whom.

Also, it is useful to have "clip animation," much like clip art, where, for example, an entire crowd of people reacts in a particular way to a character or event. We are not so concerned with their particular motions, but with the sense that "the crowd went wild" or "they respectfully parted to let her pass."

Dance is another instance where we're primarily interested in the *affect* of gesture. Music conveys emotional meaning, not literal meaning. It would be useful to be able to generate simulated dancing characters to music, by working on the level of "Now I want the dancer to convey this combination of emotions and reactions to the other dancers."

The system described here aims at allowing the building of and use of gesture on the level of affective communication. It sidesteps many difficult issues faced by systems that address robotic problems such as "Pick up the glass of water and drink it." The work described here can be plugged into such systems as a modifying filter, but is independent of them.

Road Map

The structure of this section is as follows. First will come a description of some prior and related work by others. This will be followed by an outline of the basic approach, illustrated with a simple example.

After this we will describe textural gesture and the structure of the system—what the different layers and modules are and how they communicate. We will follow this with some examples of the system in use. This section will conclude with a discussion of future and ongoing work.

Related Work

A number of researchers have done work complementary to that described in this section. Badler has specified movements in a goal-driven way and then fed those goals to a simulator (Badler, O'Rourke, and Kaufman 1980). Calvert sampled human figure motion and used Labanotation (a form of dance notation) to create an animated deterministic stick figure (Calvert, Chapman, and Patla 1980). Miller (1988b) has applied synthesis techniques to worms.

Waters (1987) did procedural modeling of faces. Deterministic methods for dance have been described by Yang (1988). Actor/script-based systems for crowds (flocks and herds) were first described by Reynolds (1987). Goal-based movement was described by Badler, O'Rourke, and Kaufman (1980).

Physically based systems for jointed motion are very powerful, although they can be difficult to control and specify (Girard and Maciejewski 1985). Fusco and Tice (1993) take a sampling approach, digitizing sequences of actual human movement and using those to drive animated figures.

Basic Notions

The basic idea is that often we don't care what a gesture is actually doing, so long as it is conveying particular emotional information. For example, when someone is explaining something, his or her arms will wave about in a particular way. Different people, and the same people in different emotional states, will do this in a way that can be defined statistically. Those watching do not care exactly where the hands or arms of the speaker are at any given moment. Instead, they watch the rhythm, the range of motion, the average posture, and the degree of regularity or irregularity of movement.

Our approach is to create a nondeterministic "texture" as a synthesis of scalar control parameters and to use these parameters to drive a motion description subsystem.

Stochastic Control of Gesture

The key innovation is the use of stochastic controls to specify gestural affect. But this needs to be placed in an appropriate system to be useful. This section consists of two parts. First we will describe stochastically defined gestures and then the system in which this technique is embedded.

Any animatable character has some N input parameters that control its motion. At any moment, the character can be thought of as residing at a point in an

N-dimensional unit cube. Each dimension spans the lowest to the highest value of one parameter.

We define a *textural gesture* as a stochastically defined path of motion within this cube that has constant statistical properties. Many gestures of a human figure can be defined by assigning to each joint angle a value for the triple `[center,amplitude,frequency]` and using the following procedure at the `time` of each frame:

```
center + amplitude * noise (frequency * time)
```

This procedure was used for most of the gestures in this section. It allows control over average position and frequency of undulation, while conveying a "natural" quality to all motion. This procedure has a constant statistical quality over time and therefore can be thought of as a fixed point in a gesture space induced over the *N*-cube space.

Several specific and interesting relationships came out of playing with these parameters. For example, I found that natural arm gestures generally resulted when elbow joints were moved with twice the frequency of shoulder joints. Varying too far from this ratio produced unnatural and artificial-looking gestures. This might be related to the fact that the weight of the entire arm is approximately twice the weight of the lower arm alone. The output of these procedures is combined via linear blending and fed into a kinematic model. In this way the single gestural "points" of the induced gesture space are used to traverse convex regions in this space.

The system provides the user with a general-purpose interpreted language for defining new gesture textures, in the spirit of Stephenson (1992). Surprisingly, almost all gestures built using the system could be defined by linearly transformed *noise* of the various joint angles. For example, the bottom row of Figure 12.27 shows a transition from a "fencing" gesture to a "conducting" gesture. This illustrates sliding along a line within the induced gesture space.

The System

The texturally defined parameters feed into "scene description modules" (SDMs). An SDM can represent an animated character, a group of animated characters, or some physical object(s) in the scene. Each SDM knows about kinematics, static constraints, dynamics, and so on (whereas the texture module does not). This separation allows a very simple, high-level control of emotive qualities and makes those qualities very easy to modify. An SDM can take as input a set of scalar parameters and generally outputs scene transformations (matrices) and geometry to be rendered. Also, one SDM can be dependent on the matrices computed by another. For clarity

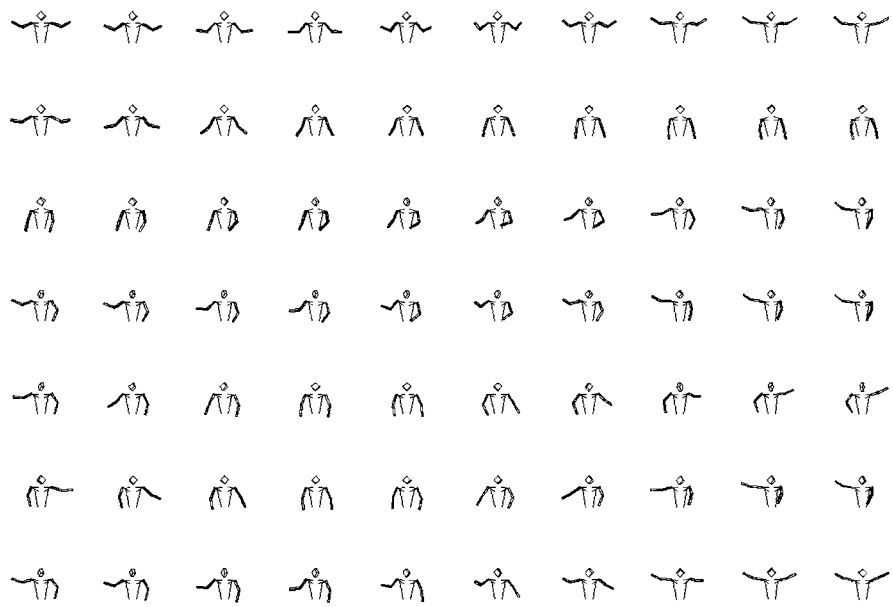


FIGURE 12.27 Gesturing man.

in this section, we will define a *parameter* as a scalar value that controls the movement of an SDM and a *gesture* as a procedure that outputs varying values over time of a set of scalar parameters that feed into an SDM.

The system evaluates a frame of animation in several layers. At the top is a goal determination module (GDM). The GDM is responsible for determining, at each frame of animation, a weight for each gesture.

Below this we have the stochastic procedures for creating individual gestures. A gesture g is invoked if its weight is nonzero. Each gesture creates values over time for some set of parameters, as described in the previous section.

Once all gestures have been evaluated, we perform for each parameter a weighted sum of its values within each gesture for which it is defined. All parameter values are then normalized by their respective accumulated weights.

Finally, all SDMs are run for this frame. The transformation matrices produced by the various SDMs are available to an analyzer, which uses them to produce scalar variables that can be fed back into the GDM. For example, as character B physically

approaches character A, the analyzer can evaluate their mutual distance and feed this back to the GDM to influence character A to increase the weight of a gesture that conveys a sense of, say, "being sad" or "waving to character B."

EXAMPLES

Figure 12.27 shows a gesticulating man at various points in his gesture space. The top row shows him at a fixed "conducting" gesture point. If held at this point, he will continue to convey exactly the same affect forever, while never actually repeating the same movement twice. The second row shows a transition between this point and a "sad" gesture point—downcast with relatively slow movement, sloping shoulders, and low energy.

The third row shows a linear transition to a "fencing" gesture point. This point is held throughout the fourth row. In the fifth and sixth rows the man switches to fencing the other way and then back again. Finally, in the bottom row he transitions back to the original gesture point.

One important thing to note is that every point in the transition between gestures produces reasonable motion. This is logical, since the statistical properties of motion during these transitions are still under tight control.

A reasonable use to this approach would be to do statistical analysis/resynthesis of human motion. This would involve analyzing statistics from real human figure motion and turning these into sum-of-noise descriptions. These gesture "points" would then be added to the system.

TEXTURE FOR FACIAL MOVEMENT

In this section we apply texture principles to the interactive animation of facial expression. Here the problems being addressed are "How do we make an embodied autonomous agent react with appropriate facial expression, without resorting to repetitive prebuilt animations?" and "How do we mix and transition between facial expressions to visually represent shifting moods and attitudes?"

By building up facial expression from component movements, and by approaching combinations of these movements as a texturing problem, it is possible to use controlled *noise* to create convincing impressions of responsiveness and mood.

This section is structured as follows. After giving a background and discussing some related work, we will introduce a simple layered movement model. Then we will describe the mechanism that allows animators to build successive abstractions.

This is followed by examples of the use of controlled *noise* to build up elements of autonomous facial movement. The section concludes with a discussion of some future work.

Background

Much human communication is conveyed by facial expression (Faigin 1990). One of the limitations of computer-human interfaces is their inability to convey the subtleties we take for granted in face-to-face communication. This concept has been well described in speculative fiction on the subject (Stephenson 1992). Toward this end, Parke (1982) and others have made good use of the Facial Action Coding System (FACS) (Ekman and Friesen 1978) for designing facially expressive automata, and there has been considerable work on computer-generated facial animation (Parke and Waters 1996).

Here we focus in particular on using procedural texture to convey some of the rich, time-varying facial expressions that people generally expect of each other. Aggressive, endearing, or otherwise idiosyncratic movements of facial expression convey a lot of our sense of another person's individuality. One inspiration for capturing this individuality can be found in successful animated characters.

For example, the hugely successful and endearing character of Gromit in Nick Park's animation (Park 1993) consistently reacts to events first with internal expressions of instinctive surprise or worry and then, a beat later, with some expression of higher judgment: disgust, realization, suspicion, and so on. Gromit's reactions become the audience's point of view. This identification creates an emotional payoff that draws in the audience (Park 1996). It would be good for an interactive character to be able to convey the same sense of a compelling emotional point of view, and to react with appropriate dynamic facial expression, without resorting to predefined expressions or to repetitive prebuilt animations.

To approach this with procedural textures, we build on Perlin and Goldberg (1996), using a parallel layered approach. The key is to allow the author of the animated agent to relate lower-level facial movements to higher-level moods and intentions, through controlled procedural textures. The animator specifies time-varying linear combinations and overlays of facial movements. Each such combination becomes a new, derived set of facial movements. In a later subsection we will show how to use controlled *noise* (Perlin 1985) to introduce controllable autonomous motion, and we will show a number of examples of autonomous facial animation built with this texturing approach.

Related Work

Facial movement synthesis by linear motion blending has been around for quite some time. In addition to Parke's work, this basic approach was also used by DeGraf and Wahrman (1988) to help drive an interactive puppeteered face.

Kalra et al. (1991) proposed an abstraction model for building facial animation, for combining vocal articulation with emotion, and for facilitating specification of dialogs in which characters could converse while conveying facial emotion. The model consisted of five abstraction layers:

1. Abstract muscles
2. Parameters built from these abstract muscles
3. Poses built by mixing parameters
4. Sequences and attack/sustain/release envelopes of poses
5. High-level scripts

Within this structure, the facial expression textures that we will describe largely contribute between levels 2 and 3 of Kalra's formalism, by providing a mechanism for combining multiple time-varying layers of successively abstracted facial expression.

Brooks (1986) developed autonomous robots having simultaneous different semantic levels of movement and control. In his *subsumption architecture,* longer-term goals were placed at higher levels, and immediate goals (such as "don't fall over") were placed at lower levels. When necessary, the lower-level controls could temporarily override higher-level activities.

Using the basic noise-based procedural texture tools of Perlin (1985), including *bias* and *gain* controls, I developed a responsive real-time dancer whose individual actions were procedurally defined (Perlin 1995). Actions could be layered and smoothly blended together to create convincing responsive body movement, which conveyed varying moods and attitudes. Athomas Goldberg and I later extended this work (Perlin and Goldberg 1996) to create characters that used a more advanced layering system for multiple levels of responsive motion. This continuous movement model was controlled by a discrete stochastic decision system, which allowed authors to create characters that would make choices based on dynamic moods, personalities, and social interactions, both with each other and with human participants.

The Movement Model

Assume that a face model consists of a set of vertices, together with some surface representation built on those vertices. We can build a component movement as a linear displacement of some subset of vertices. Thus, each movement is a list of $[i, \underline{x}]$ pairs, where i is a vertex identifier and $\underline{x}$ is the displacement vector for that vertex. We add additional movements for the axes of rigid head rotation, which are applied after all vertices have been displaced.

Given K component movements, we can give a state vector for the facial expression, consisting of linear combinations of its component movements. I used this approach to build a face that was as simple as possible, including only vertices that were absolutely necessary (Figure 12.28).

The goal was to make a model that would be used for working out a facial expression–texture vocabulary. Then the small number of vertices of this model could subsequently be used to drive movement of more elaborate facial geometries.

The face model used throughout this section contains fewer than 80 vertices, including the hair and shoulders. It requires no 3D graphical support and runs as an interactive Java applet on the Web (Perlin 1997).

The basic component movements included are

- Left/right eyebrows
- Left/right upper eyelids
- Left/right lower eyelids
- Horizontal/vertical eye gaze

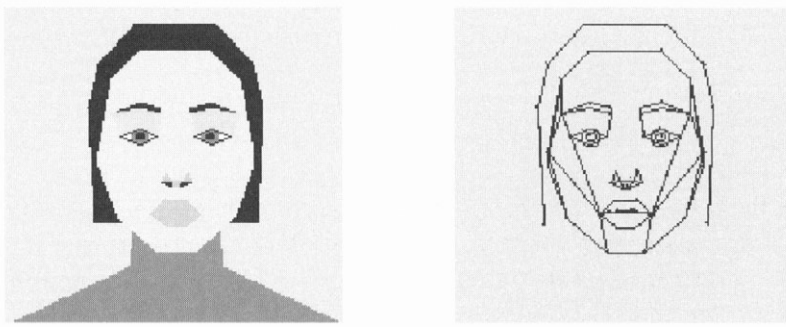


FIGURE 12.28 The movement model.

- Left/right sneer
- Left/right smile
- Mouth open
- Mouth narrowed
- Head rotation

In this chapter we will refer to each of these component movements by the following respective names:

- `BROW_L, BROW_R`
- `WINK_L, WINK_R`
- `LLID_L, LLID_R`
- `EYES_R, EYES_UP`
- `SNEER_L, SNEER_R`
- `SMILE_L, SMILE_R`
- `AHH`
- `OOH`
- `TURN, NOD`

Internally, each component movement is represented by a K-dimensional basis vector, each of which has a value of 1 in some dimension j and a value of 0 in all other dimensions. For example:

```
BROW_L = (1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
BROW_R = (0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
```

The state space for the face consists of all linear combinations of these basis vectors. Commonly used combinations can be created rather easily. For example:

```
BROWS = BROW_L + BROW_R
BLINK = WINK_L + WINK_R
SNEER = SNEER_L + SNEER_R
```

Figure 12.29 shows some simple combinations of component movements. Figure 12.30 illustrates simple movement combinations and summation of component movements, first in wire frame and then shaded.

Each component movement also defines its opposite. For example, `(-1 * SMILE)` creates a frown. By convention, the face is modeled in a neutral expression, with the

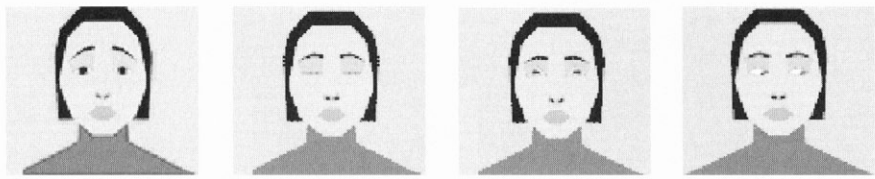


FIGURE 12.29 Simple component movement combinations.

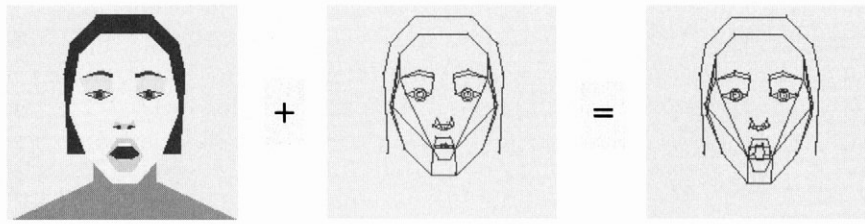


FIGURE 12.30 Summation of simple component movements in wire frame and shaded.

mouth half open, and the component movements are defined so that a range of −1 to +1 will produce a full range of natural-looking expressions.

Clearly, this is a simplified and incomplete model, sufficient just for the current purposes. The model is flexible enough to allow building textural facial expressions, but simple enough that it could be worked with quickly and easily for experiments. For experimental purposes, 16 component movements are just sufficient to allow an emotionally expressive face to be built.

Movements absent from this set include the ability to puff out or suck in the cheeks, to stick out or wave the tongue, to thrust the lower jaw forward or back, to tilt the head to the side, to displace the entire mouth sideways, as well as others. These should certainly be included in a complete facial model, which could either

retain our simple abstracted geometry or else drive a nonlinear muscle/skin model such as that of Lee, Redner, and Uselton (1985) or Chadwick, Haumann, and Parent (1989).

Movement Layering

The lowest-level abstraction is built on top of the primitive movements, using them as a vocabulary. We extend the model for mixing and layering that was described in Perlin (1995) and Perlin and Goldberg (1996). As in that model, movement is divided into *layers*. Each layer is a collection of related actions that compete for control of the same movements. Within a layer, one or more actions are always active. When a particular action A_i in a layer is activated, then the weight W_i for that action smoothly rises from zero up to one, and simultaneously the weights of all other actions in the layer descend smoothly down to zero, with a default transition time of one second.

Each action modulates the value of a set of movements (usually a fairly small set). Action A_i produces a mapping from *Time* to a list of *ComponentMovement, Value* pairs:

$$A_i : Time \text{ -> } ((D_1, V_1), \ldots, (D_k, V_k))$$

where each V_j is a scalar-valued time-varying function. In this way, an action influences one or more movements toward particular values. $A_i(Time)[D]$ refers to the value in action A_i at *Time* for component movement D.

Generally, more than one action is occurring within a layer. The total effect upon component movement D of all actions that influence it is given by the weighted sum

$$\sum_i A_i(Time)[D] * W_i(Time)$$

The total effect, or *opacity,* upon D from this group is given by

$$\sum_i W_i(Time)$$

All the groups run simultaneously. At any given moment, at least one action is running in each group. As in an optical compositing model, the groups are layered back to front. For each movement D, if the cumulative weight of a *Layer* at some *Time* is $opacity(Layer)(Time)$, then the results of all previous group influences on D are multiplied by $1 - opacity(Layer)(Time)$.

The Bottom-Level Movement Vocabulary

At the bottom level are actions that simply pose the face or else make simple movements. This level has separate layers for head position, facial expression, and mouth position. The following are some actions in each layer. All are simple poses, except for `headroll`, `headbob`, and `headshake`. The latter two are controlled by *noise*.

From the head position layer:

- `action headback { (TURN, -0.2) ), (NOD, -0.5) }`
- `action headbob { (TURN, 0), (NOD, noise(2*Time)) }`
- `action headdown { N(NOD, 1) }`
- `action headroll { (TURN, cos(Time) ), (NOD, sin(Time)) }`
- `action headshake { (TURN, noise(2*Time)), (NOD, 0) }`

From the facial expression layer:

- `action angry { (BROWS, -1), (SMILE, -.8) }`
- `action distrust { (BROW_L, -1), (LLID_L, .5), (WINK_L, .5) }`
- `action nono { (AHH, -.6), (OOH, .6), (BROWS, -1), (BLINK, .1) }`
- `action sad { (AHH, -1), (BROWS, 1), (SMILE, -.8) }`
- `action smile { (BLINK, .4), (LLIDS, .5), (BROWS, 0), (SMILE, 1), (OOH, -.6) }`
- `action sneeze { (AHH, -1), (BLINK, 1), (BROWS, 1), (LLIDS, .7), (SNEER, .7) }`
- `action surprised { (AHH, .1), (BROWS, 1), (BLINK, -.5), (LLIDS, 1) }`
- `action suspicious { (AHH, -.9), (BLINK, .5), (BROW_R, -1.2), (BROW_L, -.5), (LLIDS, 1.1) }`

From the mouth position layer:

- `action say_a { (AHH, .1), (OOH, 0) }`
- `action say_e { (AHH, -.6), (OOH, -.2) }`
- `action say_f { (AHH, -.9), (OOH, -.2), (SNEER, .2) }`
- `action say_o { (AHH, -.1), (OOH, .3) }`
- `action say_r { (AHH, -.6), (OOH, .2) }`

- `action say_u { (AHH, -.6), (OOH, .6) }`
- `action say_y { (AHH, -.7), (OOH, -.3) }`

Figure 12.31 shows the results of some pose actions.

Painting with Actions

Up to this point, the model mimics that of Perlin and Goldberg (1996), by allowing simple layered combinations of the primitive movements. It would be useful to go beyond this, treating an action as a texture. For example, the actions defined earlier can place the face into useful positions, and the layered blending mechanism will make smooth transitions between those positions. Yet the face will appear to move in a fairly lifeless and mechanical way.

To improve on this, we would like to mix some coherent jitter into many component movements, tuned to match the subtle shifting that real faces do. This is done first by defining an action that creates the jitter and then by "mixing in" amounts of this action as a transparent wash on top of the preexisting motion.

More generally, an action is a time-varying function of some set of movements. This action itself can be viewed as a primitive, which can be added and mixed. Consider the analogy with painting. An artist begins with a set of paints, each having a discrete color. The artist then creates a palette by mixing these colors to get new colors. The artist can then use each of these new colors without needing to go back to the original color set. In digital paint systems, the artist can do this with textures as well. The artist can create a texture and then paint with it, as though painting with a custom-designed textured brush.

Similarly, an action provides a way to encapsulate a time-varying function of movements and to treat this time-varying function as though it were itself a primitive movement. If noise-based variation in the action definition creates some motion "texture," then any uses of this action will reflect that texture.

Here is a more sophisticated example, which contains time-varying behavior:

```
action sneeze {
    Ah = timeCurve((0,0),(1,1),(1.5,0));
    Choo = timeCurve((1.2,0),(1.5,1),(2,0));
    ( sneeze, Ah),
    ( headup, Ah / 3),
    ( nono, Choo),
    ( headshake, Choo / 2),
    ( headdown, Choo / 2)
}
```

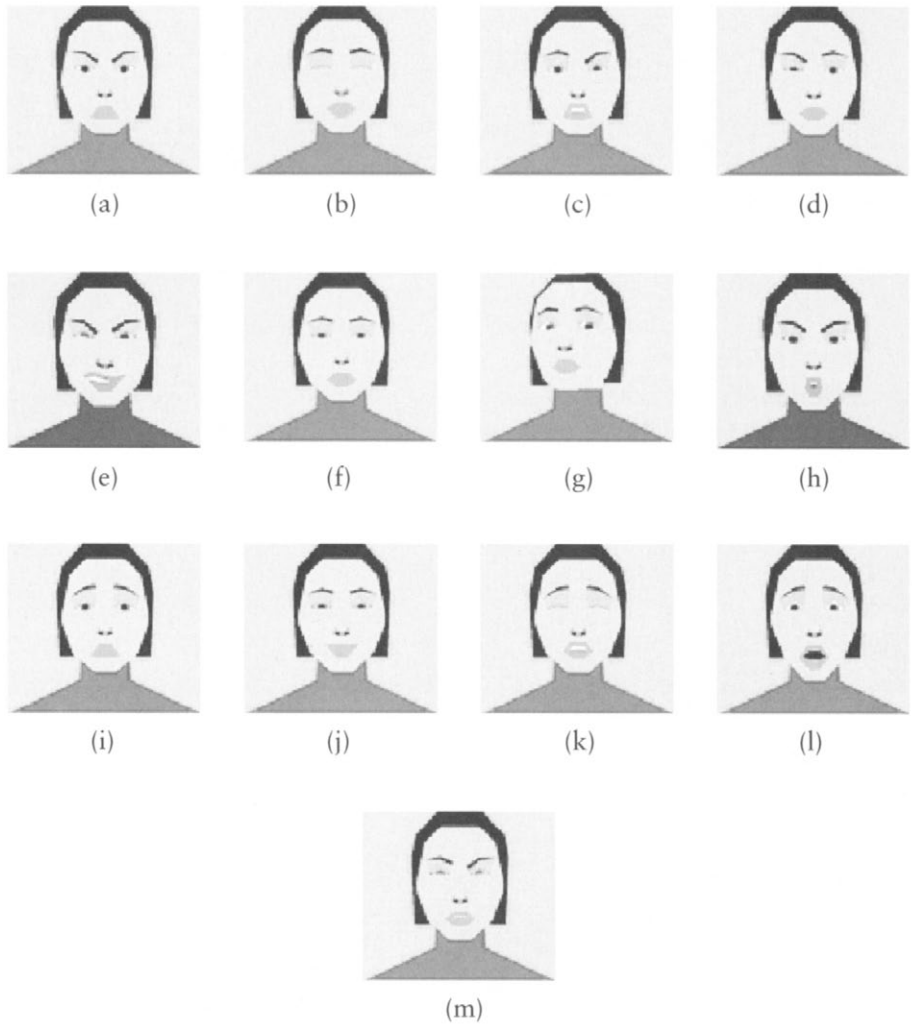


FIGURE 12.31 Results of pose actions: (a) angry; (b) daydreaming; (c) disgusted; (d) distrustful; (e) fiendish; (f) haughty; (g) head back; (h) disapproving; (i) sad; (j) smiling; (k) sneezing; (l) surprised; (m) suspicious.

where `timeCurve(...)` is a function that interpolates a smooth spline between a given sequence of `(time,value)` pairs, based on elapsed time since the onset of the action.

The animator does not need to know the details of the lower-level definitions for `sneeze`, `headup`, and so on, which frees the animator to concentrate on the details that are important at this level: tuning the time-varying behavior of the sneeze. Note that in this example the derived action inherits, via `headshake`, a realistic nondeterministic shaking during the `Choo` phase of the sneeze. The animator gets this nondeterminism for free.

Using *noise* in Movement

A number of examples will illustrate the use of *noise* in layering facial movement.

Blinking

Noise is used to trigger blinking with controlled irregularity. After each blink, we wait for one second. Then we start to check the value of a time-varying *noise*, which ranges smoothly from −1.0 to 1.0. When the value of this *noise* exceeds 0.1, we trigger another blink:

```
action blink {
  if (Time X blink.Time + 1 and noise(Time) X 0.1)
    blink.Time = Time
    (BLINK, 1) // set BLINK on this frame
  }
```

Small Constant Head Movements

As we said earlier, we would like to mix some coherent *noise* into the movement, tuned to match the subtle shifting that real faces do. To do this in a way that works with all other movement, we overlay a transparent layer of random movement over the activities of the bottom abstraction. First we add an action into the bottom abstraction that creates random movements:

```
action jitter {
    T = noise(Time);
  add (BROWS, T);
  add (LLIDS, T);
  add (EYES_R, T/2);
  add (EYES_UP, T/2);
  add (TURN, noise(Time + 10));
  add (NOD, noise(Time + 20));
}
```

Then we add an action to the next level of abstraction that creates a transparent mix-in of the lower-level jitter:

```
action jitter { jitter 0.3 }
```

The effect is subtle, yet essential. It gives a sense of life and movement to whatever the face is doing.

Both blinking and jitter are controlled by *noise*. This allows us to make movements that are unpredictable to the observer and yet that have tunable statistics that match users' expectations.

Simulated Talking

We found that we could also use tuned *noise* to create a fairly realistic simulation of the mouth movements a person makes while talking. This would be useful, for example, when an agent or character is pretending to hold a conversation off in the background:

```
action talking {
  T = (1 + noise(2 * Time)) / 2; // noise between 0 and 1

    add (AHH, lerp(gain(.9, bias(.3, T)), -1, .8)); // snap open/closed
  add (OOH, noise(2 * Time + 10.5)); // vary smoothly
  add (SNEER, lerp(bias(T, .1), -.1, .6)); // the "f" phoneme
}
```

In this implementation, we first define a *noise* process that has a frequency of two beats per second, with values in the range 0 to 1. We also use *bias*() and *gain*() as defined in Perlin (1985) to shape the *noise*, as well as linear interpolation, defined by

$$lerp(t, a, b) = a + t(b - a)$$

We use these tools to do several things simultaneously:

- Snap the mouth open and closed fairly crisply (the `gain(.9, bias(.3, T))` takes care of this).
- Smoothly make the mouth wider and narrower.
- Occasionally show the upper teeth via a `SNEER`. This gives the visual impression of an occasional *f* or *v* sound.

Once the parameters have been tuned, the result is surprisingly convincing. We use a derived abstraction to smoothly turn this talking action on and off, as when two characters are taking turns during a conversation.

Searching

We can also create a lifelike motion to make the agent appear to be addressing an audience. When people do this, they tend to look at particular people in their audience in succession, generally holding each person's gaze for a beat before moving on. We can simulate this with *noise,* by making the head and gaze follow a controlled random moving target. First we create a target that darts smoothly around, lingering for a moment in each place:

```
T = Time/3 + .3 * sin(2*PI * Time/3);    // make time undulate
Target = (noise(T), noise(T + 10) / 2); // choose moving target
```

Then we direct the `TURN`, `NOD`, `EYES_R`, and `EYES_UP` movements to follow this target point.

The first line in the previous code undulates the domain of the *noise* so that the target point will linger at each location. This is a special case of expressions having the form $X/S + A * \sin(2*\pi*F/S)$, with A in the range [0..1], which creates a monotonically increasing function that undulates regularly between slow and fast movement, with an average slope of 1 /S. In particular, the function becomes slow every S units on its domain.

By feeding such an undulating function into *noise,* we have specified a smoothly moving point that slows to a near stop at a different random target point every three seconds. We have found this to be quite effective together with the talking simulation, to create a realistic impression of a person addressing an audience.

Laughing

Here we create a laugh motion that starts by rearing the head up momentarily, and then settles into a steady state of bobbing and shaking the head while modulating an open *o* sound with a *noise* function:

```
action laugh {
  ( headshake , 0.3 ),
  ( headbob , 0.3 ),
  ( headup , timeCurve( (0,0) , (.5,.5) , (1,0) ) ),
  ( smile , 1 ),
  ( say_o , 0.8 + noise(Time) )
}
```

We can then mix this in, as a fleeting action, by deriving a higher-level laugh. This more abstracted laugh leaves the face in persistent smiling mood, until some other response causes the laugh action to be turned off:

```
action laugh {
  ( laugh , timeCurve( (0,0) , (.5,1) , (2,0) ) ),
  ( smile , 1 )
}
```

Same Action in Different Abstractions

We now describe several more actions that have been implemented at multiple semantic levels, within successively derived abstractions.

Wink

At the lowest level, we can express a wink as just the closing of one eye:

```
action wink { ( WINK_L , 1 ) }
```

At the next level we can mix in a sly or distrustful expression with the head slightly turned to one side:

```
action wink {
H = timeCurve( (0,0),(.5,.4),(.6,.5),(.9,.5),(1,.4),(3,0) );
W = gain(.999, bias(.1, 2 * H));
( wink , W ),
( distrust , H ),
( headright , 0.2 * H )
}
```

This has consistent but different effects when it is invoked while the face is in different moods: smiling, haughty, distrustful (Figure 12.32).

Note also that these haughty and distrustful facial expressions are more effective than were their lower-level equivalents in Figure 12.29. At this higher level of abstraction they have been combined with the most effective head positions. Because head position and facial expression have been mixed within a higher abstraction layer, each is correctly modulated by the wink.

Talking in Different Moods

Finally, we have created various combinations of the autonomous talking process overlaid with different moods. The movements of the face can be seen interactively at *mrl.nyu.edu/perlin/facedemo*.

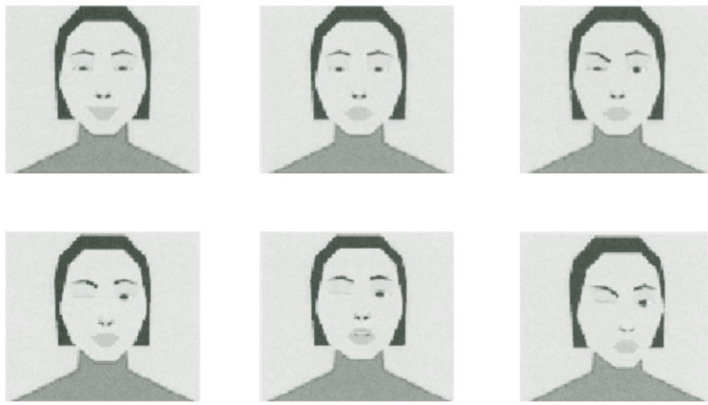


FIGURE 12.32 Various moods.

What Next?

Currently, the interface for creating individual actions is entirely script based, although I use simple drawing widgets for defining time-varying splines. There is ongoing work to create a graphic user interface, so that animators can move sliders interactively to modify component movements and mix-in weights.

Beyond this, I'm planning to extend the paint-mixing metaphor for blending actions. Within any abstraction, the animator will see a palette of small animated faces that represent the primitives available to that abstraction. The animator can then blend these to modify an action, using a time-varying brush.

I'm also continuing to increase the vocabulary of movements, working mostly from direct observation. For example, one movement to add is amused disbelief, which combines eyes looking up, smiling, and head shaking from side to side. Another is "You want this, don't you?" which combines head turned to the side, eyes looking at you, eyes wide upon, and eyebrows up. A third is puzzlement, which combines head tilted to the side (requiring an additional degree of freedom), the lower lids up, and one eyebrow raised.

In near future work, I'd like to categorize idioms that consist of mixing of lower and higher levels. One goal is to implement a Gromit-like character by the use of these mixed-level idioms. For example, the character can raise one eyebrow and then shake its head a beat later. A test of this would be to attempt to implement some of Gromit's basic emotional reaction vocabulary.

CONCLUSION

Since much of this chapter was first written, a huge revolution in procedural texturing and modeling has been taking place. Just in the last year, we have seen enormous strides in the capabilities and programmability of graphics accelerator boards. This coming SIGGRAPH (2002) will mark the first year that graphics accelerator companies will be making truly general-purpose graphics hardware–level programming available to high-level ("C style") programmers, so that their algorithms can be compiled down to accelerated and highly parallel pixel shaders.

This year, these hardware pixel shader capabilities still provide only SIMD (single instruction, multiple data) parallelism, which means that many of the techniques I began to teach about in 1984, which make much use of data-based branching and looping, are not yet available down at this fastest level.

But the gauntlet has been flung. Each year in this decade is going to see great leaps on this hardware accelerated level of capability, and at some point in the next few years we'll see high-level programmable MIMD (multiple instruction, multiple data) parallelism. At that point we will truly have real-time procedural texturing and modeling, and the world I was trying to give people a glimpse into 18 years ago will finally have arrived—a world, in the immortal words of Lance Williams, "limited only by your imagination."