7

PROCEDURAL MODELING OF GASES

DAVID S. EBERT

INTRODUCTION

This chapter presents a framework for volumetric procedural modeling and texturing. Volumetric procedural models are a general class of procedural techniques that are great for modeling natural phenomena. Most graphics applications use surface-based models of objects; however, these models are not sufficient to effectively capture the intricate space-filling characteristics of many natural phenomena, such as water, fire, smoke, steam, clouds, and other gaseous phenomena. Volume models are also used extensively for modeling fur and other "soft" objects.

Volumetric procedural models use three-dimensional volume density functions (*vdf*(*x*,*y*,*z*)) that define the model density (or opacity) of a continuous three-dimensional space. Volume density functions (vdf's) are the natural extension of solid texturing (Perlin 1985) to describe the actual geometry of objects. I have used them for modeling and animating gases such as steam, fog, smoke, and clouds (Ebert, Carlson, and Parent 1994; Ebert 1991; Ebert and Parent 1990; Ebert, Ebert, and Boyer 1990; Ebert et al. 1997). Hypertextures (Perlin and Hoffert 1989), described by Ken Perlin in Chapter 12, and Inakage's flames (Inakage 1991) are other examples of the use of volume density functions.

This chapter will focus on the use of volumetric procedural models for creating realistic images of gases, specifically smoke, steam, and fog. I will use the term *gas* to encompass gas and particulate volumes, both of which are governed by light-scattering models for small particles. Atmospheric attenuation and lighting models are robust enough to encompass both types of volumes and produce visually correct results.

As in the preceding chapters, the procedures in this chapter will make use of the stochastic functions `noise( )` and `turbulence( )`. I will give a simple implementation of the `noise( )` and `turbulence( )` functions used in my system.

This chapter first summarizes previous approaches to modeling gases, and then presents a brief description of my volume ray-tracing system for gases and several approaches for using graphics hardware to implement these effects. The concept of three-dimensional "solid spaces" is introduced next to stress the importance of the relationship of procedural (function/texture) space to object and screen space. Finally, it concludes with a detailed description of how to create still images of gases.

PREVIOUS APPROACHES TO MODELING GASES

Attempts to model gases in computer graphics started in the late 1970s. Since that time, there have been many different approaches. These can be categorized as techniques for modeling the geometry of gases and techniques for rendering scenes with gases and atmospheric effects.

There have been several approaches to modeling the geometry of gases. Some authors use a constant density medium (Klassen 1987; Nishita, Miyawaki, and Nakamae 1987) but do allow different layers of constant densities. Still, only very limited geometries for gases can be modeled. Voss and Musgrave use fractals to create realistic clouds and fog effects (Voss 1983; Musgrave 1990). Max (1986) uses height fields for simulating light diffusion effects, and Kajiya uses a physically based model for modeling clouds (Kajiya and Von Herzen 1984). Gardner (1985, 1990) has produced extremely realistic images of clouds by using Fourier synthesis to control the transparency of hollow ellipsoids. The main disadvantage of this approach is that it is not a true three-dimensional geometric model for the clouds.

I have developed several approaches for modeling gases based on volume density functions (Ebert 1991; Ebert and Parent 1990; Ebert, Ebert, and Boyer 1990; Ebert, Boyer, and Roble 1989). These models are true three-dimensional models for the geometry of gases and provide more realistic results than previous techniques. Stam and Fiume (1991, 1993, 1995) also use a three-dimensional geometric model for gases. This model uses "fuzzy blobbies," which are similar to volumetric metaballs and particle systems, for the geometric model of the gases. Stam and Fedkiw have extended this work to use physically based Navier-Stokes solutions for modeling gases and have achieved very realistic effects (Stam 1999; Fedkiw, Stam, and Jensen 2001). Sakas (1993) uses spectral synthesis to define three-dimensional geometric models for gases. Many authors have used a variety of techniques for the detailed modeling and real-time approximation of clouds, which is described in Chapter 9.

The rendering of scenes containing clouds, fog, atmospheric dispersion effects, and other gaseous phenomena has also been an area of active research in computer graphics. Several papers describe atmospheric dispersion effects (Willis 1987; Nishita, Miyawaki, and Nakamae 1987; Rushmeier and Torrance 1987; Musgrave 1990), while others cover the illumination of these gaseous phenomena in detail (Blinn 1982a; Kajiya and Von Herzen 1984; Max 1986; Klassen 1987; Ebert and Parent 1990). Most authors use a low-albedo reflection model, while a few (Blinn 1982a; Kajiya and Von Herzen 1984; Rushmeier and Torrance 1987; Max 1994; Nishita, Nakamae, and Dobashi 1996; Wann Jensen and Christensen 1998; Fedkiw, Stam, and Wann Jensen 2001) discuss the implementation of a high-albedo model. (A low-albedo reflectance model assumes that secondary scattering effects are negligible, while a high-albedo illumination model calculates the secondary and higher-order scattering effects.) There has also been considerable work in the past several years in developing interactive rendering techniques for gases and clouds, described in Chapter 10.

THE RENDERING SYSTEM

For true three-dimensional images and animations of gases, volume rendering must be performed. Any volumetric rendering system, such as the systems described by Perlin and Hoffert (1989) and Kajiya and Von Herzen (1984), or approximated volume-rendering system can be used, provided that you can specify procedures to define the density/opacity of each volume element for the gas. I will briefly discuss my rendering approach, which is described in detail in Ebert and Parent (1990). This hybrid rendering system uses a fast scanline a-buffer rendering algorithm for the surface-defined objects in the scene, while volume-modeled objects are volume rendered using a per-pixel volume ray-tracing technique. The algorithm first creates the a-buffer for a scanline containing a list for each pixel of all the fragments that partially or fully cover the pixel. Then, if a volume is active for a pixel, the extent of volume rendering necessary is determined. The volume rendering is performed next, creating a-buffer fragments for the separate sections of the volumes. (It is necessary to break the volume objects into separate sections that lie in front of, in between, and behind the surface-based fragments in the scene to generate correct images.) The volume ray tracing that is used is a very simple extension to a traditional ray tracer: instead of stopping the tracing of a ray when an object is hit, the tracing actually steps through the object/volume at a defined step size and accumulates the opacity and color of each small volumetric segment of the object/volume. Volume rendering ceases once full coverage of the pixel by volume or surfaced-defined elements is achieved. Finally, these volume a-buffer fragments are sorted into the a-buffer fragment list based on their average Z-depth values, and the a-buffer fragment list is rendered to produce the final color of the pixel.

Volume-Rendering Algorithm

The volume-rendering technique used for gases in this system is similar to the one discussed in Perlin and Hoffert (1989). The ray from the eye through the pixel is traced through the defining geometry of the volume. For each increment through the volume sections, the volume density function is evaluated. The color, density, opacity, shadowing, and illumination of each sample is then calculated. The illumination and densities are accumulated based on a low-albedo illumination model for gases and atmospheric attenuation.

The basic gas volume-rendering algorithm is the following:

```
for each section of gas
  for each increment along the ray
     get color, density, & opacity of this element
     if self_shadowing
        retrieve the shadowing of this element from the
            solid shadow table
     color = calculate the illumination of the
             gas using opacity, density, and
             the appropriate model
     final_clr = final_clr + color;
     sum_density = sum_density + density;
     if( transparency < 0.01)
        stop tracing
    increment sample_pt
  create the a_buffer fragment
```

In sampling along the ray, a Monte Carlo method is used to choose the sample point to reduce aliasing artifacts. The opacity is the density obtained from evaluating the volume density function multiplied by the step size. This multiplication is necessary because in the gaseous model we are approximating an integral to calculate the opacity along the ray (Kajiya and Von Herzen 1984). The approximation used is

$$\text{opacity} = 1 - e^{-\tau \times \sum_{t_{near}}^{t_{far}} \rho(x(t), y(t), z(t)) \times \Delta t}$$

where τ is the optical depth of the material, $\rho(\ )$ is the density of the material, t_{near} is the starting point for the volume tracing, and t_{far} is the ending point. The final increment along the ray may be smaller, so its opacity is scaled proportionally (Kajiya and Kay 1989).

Illumination of Gaseous Phenomena

The system uses a low-albedo gaseous illumination model based on Kajiya and Von Herzen (1984). The phase functions that are used are sums of Henyey-Greenstein functions as described in Blinn (1982a). The illumination model is the following:

$$B = \sum_{t_{near}}^{t_{far}} e^{-\tau \times \sum_{t_{near}}^{t} \rho(x(u)y(u), z(u)) \times \Delta u} \times I \times \rho(x(t), y(t), z(t)) \times \Delta t$$

where I is

$$\sum_i I_i(x(t), y(t), z(t)) \times phase(\theta)$$

$phase(\theta)$ is the phase function, the function characterizing the total brightness of a particle as a function of the angle between the light and the eye (Blinn 1982a). $I_i(x(t),y(t),z(t))$ is the amount of light from light source I reflected from this element.

Self-shadowing of the gas is incorporated into I by attenuating the brightness of each light. An approximation for a high-albedo illumination model can also be incorporated by adding an ambient term based on the albedo of the material into I_i. This ambient term accounts for the percentage of light reflected from the element due to second- and higher-order scattering effects.

Volumetric Shadowing

Volumetric shadowing is important in obtaining accurate images. As mentioned above, self-shadowing can be incorporated into the illumination model by attenuating the brightness of each light. The simplest way to self-shadow the gas is to trace a ray from each of the volume elements to each of the lights, determining the opacity of the material along the ray using the preceding equation for opacity. This method is similar to shadowing calculations performed in ray tracing and can be very slow. My experiments have shown that ray-traced self-shadowing can account for as much as 75% to 95% of the total computation time.

To speed up shadowing calculations, a precalculated table can be used. Kajiya discusses this approach with the restriction that the light source be at infinity (Kajiya and Von Herzen 1984; Kajiya and Kay 1989). I have extended this approach to remove this restriction. Using my technique, the light source may even be inside the volume. This shadow-table-based technique can improve performance by a factor of 10–15 over the ray-traced shadowing technique. A complete description of this shadowing technique can be found in Ebert (1991).

The shadow table is computed once per frame. To use the shadow table when volume tracing, the location of the sample point within the shadow table is determined. This point will lie within a parallelepiped formed by eight table entries. These eight entries are trilinearly interpolated to obtain the sum of the densities between this sample point and the light. To determine the amount of light attenuation, the following formula is used.

$$\text{light\_atten} = 1 - e^{-\tau \times \text{sum\_densities} \times \text{step\_size}}$$

As mentioned above, this shadow table algorithm is much more efficient than the ray-tracing shadowing algorithm. Another benefit of this approach is the flexibility of detail on demand. If very accurate images are needed, the size of the shadow table can be increased. If the volume is very small in the image and very accurate shadows are not needed, a small resolution shadow table (e.g., 8^3) can be chosen. For most images, I use a shadow table size of 32^3 or 64^3.

Recent work in shadowing for volumetric objects provides more efficient shadow rendering. The deep shadow map approach is an extension of traditional two-dimensional texture mapping that allows shadows from semitransparent volumetric objects (Lokovic and Veach 2000) by storing a visibility function in each entry in the deep shadow map. This visibility function stores, for each depth, the fraction of light that reaches this depth.[1] Kim and Neumann (2001) have developed a hardware-accelerated opacity shadow mapping technique that is similar to 3D hardware texture-based volume rendering, using a large number (e.g., 100–500) of opacity maps to accurately calculate volumetric shadows. Kniss, Kindlmann, and Hansen (2002) have recently developed a 3D hardware texture map slicing technique that allows shadows from volumetric objects, including gases that are rendered using three-dimensional texture maps.

ALTERNATIVE RENDERING AND MODELING APPROACHES FOR GASES

There are three types of alternative rendering approaches commonly used for gases:

- Particle systems
- Billboards and imposters
- Three-dimensional hardware texture mapping

1. In actuality, a piecewise linear approximation of the function is stored.

Particle systems are most commonly used for thin gases, such as smoke. There are two problems in using particle systems for gases. The first is the complexity of computing particle self-shadowing, and the second is the computational complexity of simulating large or dense areas of gas (millions of particles may be needed for the simulation). Billboards and imposters have been effectively used for interactive cloud rendering, with some limitations imposed on the animation of the clouds and/or light sources for efficiency in interactive rendering (Dobashi et al. 2000; Harris and Lastra 2001). Three-dimensional hardware texture mapping can be used with slice-based volume rendering to simulate clouds and other dense gases (Kniss, Kindlmann, and Hansen 2002). A combination of a coarse volume representation and procedural detail, which can be rendered in the graphics processor, is used to produce convincing volumetric effects and is described in more detail in Chapter 10.

A PROCEDURAL FRAMEWORK: SOLID SPACES

This section describes a general and flexible framework for procedural techniques, called *solid spaces*. The development of this framework, its mathematical definition, and its role in procedural texturing and modeling are described below.

Development of Solid Spaces

My approach to modeling and animating gases started with work in solid texturing. Solid texturing can be viewed as creating a three-dimensional color space that surrounds the object. When the solid texture is applied to the object, it is as if the defining space is being carved away. A good example of this is using solid texturing to create objects made from wood and marble. The solid texture space defines a three-dimensional volume of wood or marble, in which the object lies.

Most of my solid texturing procedures are based on the noise and turbulence functions. This work extended to modeling gases when I was asked to produce an image of a butterfly emerging from fog or mist. Since gases are controlled by turbulent flow, it seemed natural to somehow incorporate the use of noise and turbulence functions into this modeling. My rendering system already supported solid texturing of multiple object characteristics, so the approach that I developed was to use solid textured transparency to produce layers of fog or clouds. The solid textured transparency function was, of course, based on turbulence. This approach is very similar to Gardner's approach (Gardner 1985) and has the same disadvantage of not being a true three-dimensional model, even though the solid texture procedure is defined throughout three-space. In both cases, these three-dimensional procedures are evaluated only at the surfaces of objects. To remedy this shortcoming, my next extension was to use turbulence-based procedures to define the density of three-dimensional volumes, instead of controlling the transparency of hollow surfaces.

As you can see, the idea of using three-dimensional spaces to represent object attributes such as color, transparency, and even geometry is a common theme in this progression. My system for representing object attributes using this idea is termed *solid spaces*. The solid space framework encompasses traditional solid texturing, hypertextures, and volume density functions within a unified framework.

Description of Solid Spaces

Solid spaces are three-dimensional spaces associated with an object that allow for control of an attribute of the object. For instance, in solid color texturing, described in Chapters 2 and 6, the texture space is a solid space associated with the object that defines the color of each point in the volume that the object occupies. This space can be considered to be associated with, or represent, the space of the material from which the object is created.

Solid spaces have many uses in describing object attributes. As mentioned earlier, solid spaces can be used to represent the color attributes of an object. This is very natural for objects whose color is determined from procedures defining a marble color space, as in Figure 8.2. Many authors use solid color spaces for creating realistic images of natural objects (Perlin 1985; Peachey 1985; Musgrave and Mandelbrot 1989). Often in solid texturing (using solid color spaces) there are additional solid spaces, which are combined to define the color space. For example, in most of my work in solid texturing, a noise and turbulence space is used in defining the color space. Other solid space examples include geometry (hypertextures and volume density functions), roughness (solid bump mapping), reflectivity, transparency, illumination characteristics, and shadowing of objects. Solid spaces can even be used to control the animation of objects, as will be described in the next chapter.

Mathematical Description of Solid Spaces

Solid spaces can be described simply in mathematical terms. They can be considered to be a function from three-space to *n*-space, where *n* can be any nonzero positive integer. More formally, solid spaces can be defined as the following function:

$$S(x,y,z) = F, F \in R^n, n \in 1, 2, 3, \ldots$$

Of course, the definition of the solid space can change over time; thus, time could be considered to be a fourth dimension to the solid space function. For most uses of solid spaces, S is a continuous function throughout three-space. The exception is the use of solid spaces for representing object geometries. In this case, S normally has a discontinuity at the boundary of the object. For example, in the case of implicit surfaces, S is normally continuous throughout the surface of the object, but thresholding is used to change the density value abruptly to 0 for points whose density is not within a narrow range of values that defines the surface of the object. The choice of F determines the frequencies in the resulting solid spaces and, therefore, the amount of aliasing artifacts that may appear in a final image.

GEOMETRY OF THE GASES

Now that some background material has been discussed, this section will describe detailed procedures for modeling gases. As mentioned in the introduction, the geometry of the gases is modeled using turbulent-flow-based volume density functions. The volume density functions take the location of the point in world space, find its corresponding location in the turbulence space (a three-dimensional space), and apply the turbulence function. The value returned by the turbulence function is used as the basis for the gas density and is then "shaped" to simulate the type of gas desired by using simple mathematical functions. In the discussion that follows, I will first describe my noise and turbulence functions and then describe the use of basic mathematical functions for shaping the gas. Finally, the development of several example procedures for modeling the geometry of the gases will be explored.

My Noise and Turbulence Functions

In earlier chapters of this book, detailed descriptions of noise and turbulence were discussed, including noise and turbulence functions with much better spectral characteristics. I am providing my implementations to enable the reader to reproduce the images of gases described in this chapter. If other noise implementations are used, then the gas shaping that is needed will be slightly different. (I have experimented with this.) My noise implementation uses trilinear interpolation of random numbers stored at the lattice points of a regular grid. I use a grid size of $64 \times 64 \times 64$. The 3D array is actually $65 \times 65 \times 65$ with the last column equaling the first column to make accessing entries easier (`noise[64][64][64] = noise[0][0][0]`). To implement this using 3D texture mapping hardware, you can simply create the $64 \times 64 \times 64$ table and turn the texture repetition mode to repeat. This random number lattice-based noise implementation is actually very well suited for 3D texture mapping hardware implementation, and the simple DirectX or OpenGL calls to read values from this 3D texture map will perform the noise lattice interpolation automatically.

The noise lattice is computed and written to a file using the following code:

```
// /////////////////////////////////////////////////////
//                 WRITE_NOISE.C
// This program generates a noise function file for solid texturing.
//             by David S. Ebert
// /////////////////////////////////////////////////////
#include <math.h>
#include <stdio.h>
#define SIZE 64
double drand48();
int main(int argc, char **argv )
{
  long i,j, k, ii,jj,kk;
  float noise[SIZE+1][SIZE+1][SIZE+1];
  FILE *noise_file;
  noise_file = fopen("noise.data","w");

  for (i=0; i<SIZE; i++) for
    (j=0; j<SIZE; j++) for
    (k=0; k<SIZE; k++)
       {
        noise[i][j][k] = (float)drand48( );
       }
// This is a hack, but it works. Remember this is
// only done once.
 for (i=0; i<SIZE+1; i++)
   for (j=0; j<SIZE+1; j++)
     for (k=0; k<SIZE+1; k++)
      {
       ii = (i == SIZE)? 0: i;
       jj = (j == SIZE)? 0: j;
       kk = (k == SIZE)? 0: k;
       noise[i][j][k] = noise[ii][jj][kk];
      }
   fwrite(noise,sizeof(float),(SIZE+1)*(SIZE+1)*(SIZE+1),
        noise_file);
fclose(noise_file);
}
```

To compute the noise for a point in three-space, the `calc_noise( )` function given below is called. This function replicates the noise lattice to fill the positive octant of three-space. To use this procedure, the points must be in this octant of space. I allow the user to input scale and translation factors for each object to position the object in the noise space.

The noise procedure given below, `calc_noise`, uses trilinear interpolation of the lattice point values to calculate the noise for the point. The `turbulence( )` function given below is the standard Perlin turbulence function (Perlin 1985).

```
typedef struct xyz_td
 {
  float x, y, z;
 } xyz_td;
float calc_noise( );
float turbulence( );

// ///////////////////////////////////////////////////////
//                   Calc_noise
// This is basically how the trilinear interpolation works. I
// lerp down the left front edge of the cube, then the right
// front edge of the cube(p_l, p_r). Then I lerp down the left
// back and right back edges of the cube (p_l2, p_r2). Then I
// lerp across the front face between p_l and p_r (p_face1). Then
// I lerp across the back face between p_l2 and p_r2 (p_face2).
// Now I lerp along the line between p_face1 and p_face2.
// ///////////////////////////////////////////////////////
float calc_noise(xyz_td pnt)
{
  float t1;
  float p_l,p_l2,// value lerped down left side of face 1 & face 2
       p_r,p_r2, // value lerped down left side of face 1 & face 2
       p_face1,  // value lerped across face 1 (x-y plane ceil of z)
       p_face2,  // value lerped across face 2 (x-y plane floor of z)
       p_final;  //value lerped through cube (in z)
  extern float noise[SIZE+-1][SIZE+-1][SIZE+-1];
  register int x, y, z, px, py, pz;

  px = (int)pnt.x;
  py = (int)pnt.y;
  pz = (int)pnt.z;
  x = px &(SIZE); // make sure the values are in the table
  y = py &(SIZE); // Effectively replicates table throughout space
  z = pz &(SIZE);

  t1  =     pnt.y - py;
  p_l =  noise[x][y][z+1]+t1*(noise[x][y+1][z+1]-
         noise[x][y][z+1]);
  p_r =  noise [x+1][y][z+1]+t1*(noise[x+1][y+1][z+1]-
         noise[x+1][y][z+1]);
  p_l2 = noise[x][y][z]+ t1*(noise[x][y+1][z] -
         noise[x][y][z]);
  p_r2 = noise[x+1][y][z]+ t1*(noise[x+1][y+1][z]-noise[x+1][y][z]);
  t1 = pnt.x - px;
  p_face1 = p_l + t1 * (p_r - p_l);
  p_face2 = p_l2 + t1 * (p_r2 - p_l2);
  t1 = pnt.z - pz;
  p_final = p_face2 + t1*(p_face1 - p_face2);
  return(p_final);
}
```

```
//
// //////////////////////////////////////////////////////
//              TURBULENCE
// //////////////////////////////////////////////////////
float turbulence(xyz_td pnt, float pixel_size)
{
  float t, scale;
  t=0;
  for(scale=1.0; scale >pixel_size; scale/=2.0)
    {
     pnt.x = pnt.x/scale; pnt.y = pnt.y/scale;
     pnt.z = pnt.z/scale;
     t+= calc_noise(pnt)* scale;
    }
  return(t);
}
```

Neither of these routines is optimized. Using bit-shifting operations to index into the integer lattice can optimize the noise lattice access. Precalculating a table of scale multipliers and using multiplication by reciprocals instead of division can optimize the turbulence function.

Basic Gas Shaping

Several basic mathematical functions are used to shape the geometry of the gas. The first of these is the power function. Let's look at a simple procedure for modeling a gas and see the effects of the power function, and other functions, on the resulting shape of the gas.

```
void basic_gas(xyz_td pnt, float *density,float *parms)
{
 float turb;
 int i;
 static float pow_table[POW_TABLE_SIZE];
 static int calcd=1;

 if(calcd) { calcd=0;
   for(i=POW_TABLE_SIZE-1; i>=0; i--)
      pow_table[i]=(float)pow(((double)(i))/(POW_TABLE_SIZE-1)*
                   parms[1]*2.0,(double)parms[2]);
   }
 turb =turbulence(pnt, pixel_size);
 *density =pow_table[(int)(turb*(.5*(POW_TABLE_SIZE-1)))];
}
```

This procedure takes as input the location of the point being rendered in the solid space, `pnt`, and a parameter array of floating-point numbers, `parms`. The returned value is the density of the gas. `parms[1]` is the maximum density value for the gas with a range of 0.0–1.0, and `parms[2]` is the exponent for the power function.

Figure 7.1 shows the effects of changing the power exponent, with `parms[1] = 0.57`. Clearly, the greater the exponent, the greater the contrast and definition to the gas plume shape. With the exponent at 1, there is a continuous variation in the density of the gas; with the exponent at 2, it appears to be separate individual plumes of gas. Therefore, depending on the type of gas being modeled, the appropriate exponential value can be chosen. This procedure also shows how precalculated tables can increase the efficiency of the procedures. The `pow_table[ ]` array is calculated once per image and assumes that the maximum density value, `parms[1]`, is constant for each given image. A table size of 10,000 should be sufficient for producing accurate images. This table is used to limit the number of `pow` function calls. If the following straightforward implementation were used, a power function call would be needed per volume density function evaluation:

```
*density = (float) pow((double)turb*parms[1],(double)parms[2]);
```

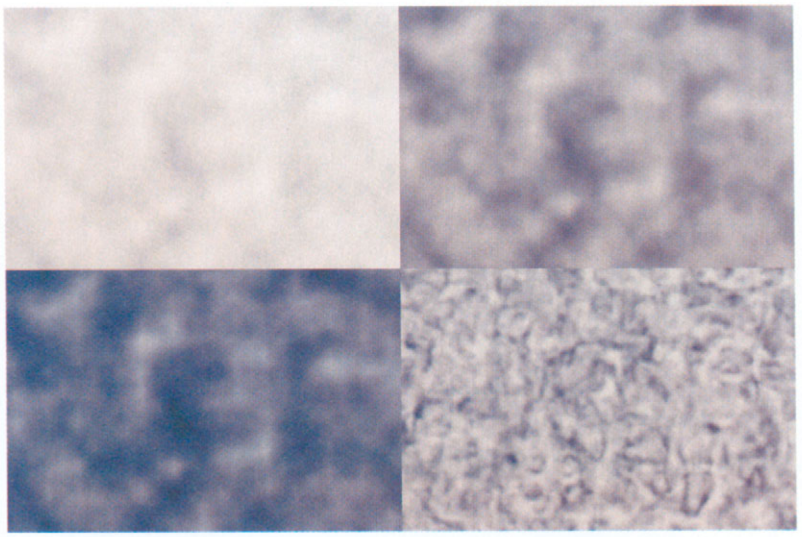

FIGURE 7.1 The effects of the power and sine function on the gas shape: (top left) a power exponent of 1; (top right) a power exponent of 2; (bottom left) a power exponent of 3; (bottom right) the sine function applied to the gas. Copyright © 1994 David S. Ebert.

Assuming an image size of 640 × 480, with 100 volume samples per pixel, the use of the precomputed table saves 30,710,000 pow function calls.

Another useful mathematical function is the sine function. Perlin (1985) uses the sine function in solid texturing to create marble, which will be described in a later section. This function can also be used in shaping gases, which can be accomplished by making the following change to the basic_gas function:

```
turb =(1.0 +sin(turbulence(pnt, pixel_size)*M_PI*5))*.5;
```

This change creates "veins" in the shape of the gas, similar to the marble veins in solid texturing. As can be seen from these examples, it is very easy to shape the gas using simple mathematical functions. The remainder of this chapter will extend this basic_gas procedure to produce more complex shapes in the gas.

Patchy Fog

The first example of still gas is patchy fog. The earlier basic_gas function can be used to produce still images of patchy fog. For nice fog, parms[1]=0.5, parms[2]= 3.0. The parms[2] value determines the "patchiness" of the fog, with lower values giving more continuous fog. parms[1] controls the denseness of the fog that is in the resulting image.

Steam Rising from a Teacup

The goal of our second example is to create a realistic image of steam rising from a teacup. The first step is to place a "slab" (Kajiya 1986) of volume gas over the teacup. (Any ray-traceable solid can be used for defining the extent of the volume.) As steam is not a very thick gas, a maximum density value of 0.57 will be used with an exponent of 6.0 for the power function. The resulting image in Figure 7.2 (left) was produced from the preceding basic_gas procedure.

The image created, however, does not look like steam rising from a teacup. First, the steam is not confined to only above and over the cup. Second, the steam's density does not decrease as it rises. These problems can be easily corrected. To solve the first problem, ramp off the density spherically from the center of the top of the tea. This will confine the steam within the radius of the cup and make the steam rise higher over the center of the cup. The following steam_slab1 procedure incorporates these changes into the basic_gas procedure:

```
void steam_slab1(xyz_td pnt, xyz_td pnt_world, float
                *density, float *parms, vol_td vol)
```

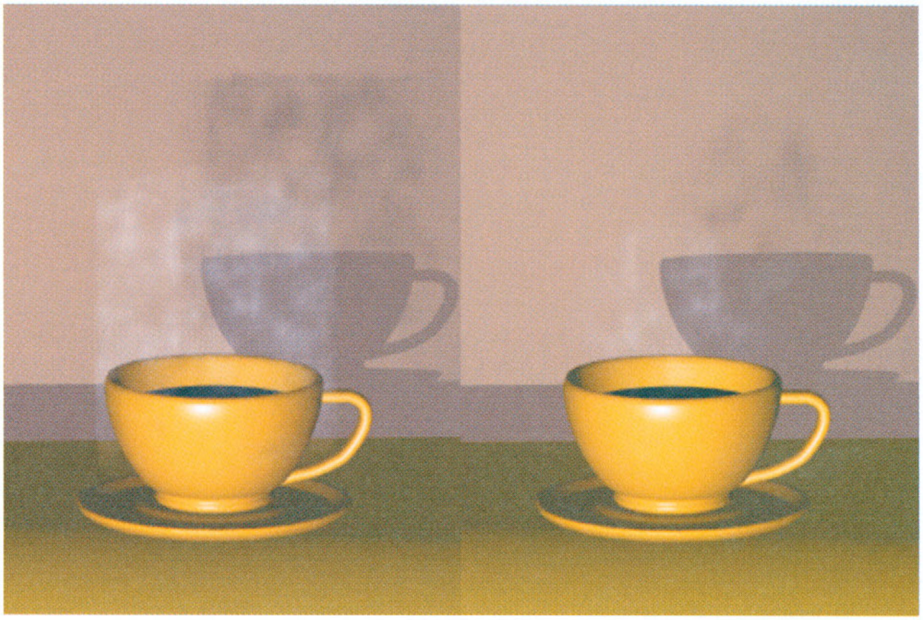

FIGURE 7.2 Preliminary steam rising from a teacup. Left: No shaping of the steam. Right: Only spherical attenuation. Copyright © 1992 David S. Ebert.

```
{
 float     turb, dist_sq,density_max;
   int
i, indx;
xyz_td diff;
static float pow_table[POW_TABLE_SIZE], ramp[RAMP_SIZE],
            offset[OFFSET_SIZE];
static int calcd=1;
if(calcd) { calcd=0;
   for(i=POW_TABLE_SIZE-1; i>=0; i--)
     pow_table[i] =
        (float)pow(((double)(i))/(POW_TABLE_SIZE-1)*
                  parms[1]* 2.0,(double)parms[2]);
   make_tables(ramp);
  }
turb = fast_turbulence(pnt, pixel_size);
*density = pow_table[(int)(turb*0.5*(POW_TABLE_SIZE-1))];

 // determine distance from center of the slab ^2.
 XYZ_SUB(diff,vol.shape.center, pnt_world);
 dist_sq = DOT_XYZ(diff,diff);
 density_max = dist_sq*vol.shape.inv_rad_sq.y;
 indx = (int) ((pnt.x+pnt.y+pnt.z)*100) & (OFFSET_SIZE -1);
 density_max += parms[3]*offset[indx];

  if(density_max >= .25) // ramp off if > 25% from center
  { // get table index 0:RAMP_SIZE-1
    i = (density_max -.25)*4/3*RAMP_SIZE;
    i=MIN(i,RAMP_SIZE-1);
    density_max = ramp[i];
    *density *=density_max;
  }
}

void make_tables(float *ramp, float *offset)
{
  int i;
  float dist;
  srand48(42);
  for(i=0; i < OFFSET_SIZE; i++)
   {
    offset[i] = (float)drand48( );
   }
 for(i = 0; i < RAMP_SIZE; i++)
  { dist =i/(RAMP_SIZE -1.0);
    ramp[i]=(cos(dist*M_PI) +1.0)/2.0;
   }
}
```

These modifications produce the more realistic image seen in Figure 7.2 (right). Two additional parameters are used in this new procedure: `pnt_world` and `vol`. `pnt_world` is the location of the point in world space; `vol` is a structure containing information on the volume being rendered. Table 7.1 clarifies the use of the various variables.

The procedure now ramps off the density spherically using a cosine falloff function. If the distance from the center squared is greater than 25%, the cosine falloff is applied. The resulting image can be seen on the right in Figure 7.2. This image is better than the one shown on the left in Figure 7.2, but still lacking.

To solve the second problem, the gas density decreasing as it rises, the density of the gas needs to be ramped off as it rises to get a more natural look. The following addition to the end of the `steam_slab1` procedure will accomplish this:

```
dist = pnt_world.y - vol.shape.center.y;
if(dist > 0.0)
  { dist = (dist +offset[indx]*.1)*vol.shape.inv_rad.y;
    if(dist > .05)
```

TABLE 7.1 VARIABLES FOR STEAM PROCEDURE

VARIABLE	DESCRIPTION
`pnt`	location of the point in the solid texture space
`pnt_world`	location of the point in world space
`density`	the value returned from the function
`parms[1]`	maximum density of the gas
`parms[2]`	exponent for the power function for gas shaping
`parms[3]`	amount of randomness to use in falloff
`parms[4]`	distance at which to start ramping off the gas density
`vol.shape.center`	center of the volume
`vol.shape.inv_rad_sq`	1/radius squared of the slab
`dist_sq`	point's distance squared from the center of the volume
`density_max`	density scaling factor based on distance squared from the center
`indx`	an index into a random number table
`offset`	a precomputed table of random numbers used to add noise to the ramp off of the density
`ramp`	a table used for cosine falloff of the density values

```
    { offset2 = (dist -.05)*1.111111;
     offset2 = 1 - (exp(offset2)-1.0)/1.718282;
     offset2 *= parms[1];
     *density *= offset;
    }
}
```

This procedure uses the e^x function to decrease the density as the gas rises. If the vertical distance above the center is greater than 5% of the total distance, the density is exponentially ramped off to 0. The result of this addition to the above procedure can be seen in Figure 7.3. As can be seen in this image, the resulting steam is very convincing. In the next chapter, animation effects using this basic steam model will be presented.

A Single Column of Smoke

The final example procedure creates a single column of rising smoke. The basis of the smoke shape is a vertical cylinder. Two observations can make the resulting

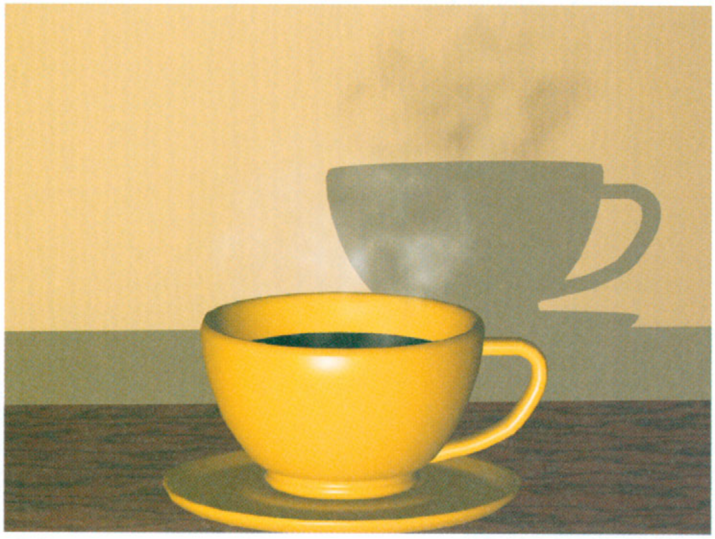

FIGURE 7.3 Final image of steam rising from a teacup, with both spherical and height density attenuation. Copyright © 1997 David S. Ebert.

image look realistic. First, smoke disperses as it rises. Second, the smoke column is initially fairly smooth, but as the smoke rises, turbulent behavior becomes the dominant characteristic of the flow. In order to reproduce these observations, turbulence is added to the cylinder's center to make the column of smoke look more natural. To simulate air currents and general turbulent effects, more turbulence is added as the height from the bottom of the smoke column increases. To simulate dispersion, the density of the gas is ramped off to zero as the gas rises. These ideas will produce a very straight column of smoke. The following additional observation will make the image more realistic: smoke tends to bend and swirl as it rises. Displacing each point by a vertical spiral (helix) creates the swirling of the smoke. The x- and z-coordinates of the point are displaced by the cosine and sine of the angle of rotation. The y-coordinate of the point is displaced by the turbulence of the point. The following procedure produces a single column of smoke based on these observations.

```
// //////////////////////////////////////////////////////////////
//                     Smoke_stream
// //////////////////////////////////////////////////////////////
// parms[1] = Maximum density value - density scaling factor
// parms[2] = height for 0 density (end of ramping it off)
// parms[3] = height to start adding turbulence
// parms[4] = height(length) for maximum turbulence;
// parms[5] = height to start ramping density off
// parms[6] = center.y
// parms[7] = speed for rising
// parms[8] = radius
// parms[9] = max radius of swirling
// ///////////////////////////////////////////////////////////

void smoke_stream(xyz_td pnt, float *density, float *parms,
                  xyz_td pnt_world, vol_td *vol)
{
 float        dist_sq;
 extern float offset[OFFSET_SIZE];
 xyz_td       diff;
 xyz_td       hel_path, new_path, direction2, center;
 double       ease( ), turb_amount, theta_swirl, cos_theta,
              sin_theta;
 static int   calcd=1;
 static float cos_theta2, sin_theta2;
static xyz_td bottom;
static double rad_sq, max_turb_length, radius, big_radius,
              st_d_ramp, d_ramp_length, end_d_ramp,
              inv_max_turb_length;
double        height, fast_turb, t_ease, path_turb, rad_sq2;
if(calcd)
  { bottom.x = 0; bottom.z = 0;
   bottom.y = parms[6];
   radius   = parms[8];
   big_radius = parms[9];
   rad_sq = radius*radius;
   max_turb_length = parms[4];
   inv_max_turb_length = 1/max_turb_length;
   st_d_ramp = parms[5];
   end_d_ramp = parms[2];
   d_ramp_length = end_d_ramp - st_d_ramp;
   theta_swirl = 45.0*M_PI/180.0; // swirling effect
   cos_theta = cos(theta_swirl);
   sin_theta = sin(theta_swirl);
   cos_theta2 = .01*cos_theta;
   sin_theta2 = .0075*sin_theta;
   calcd=0;
  }

height = pnt_world.y - bottom.y + fast_noise(pnt)*radius;
// We don't want smoke below the bottom of the column
if(height < 0)
  { *density =0; return;}
height -= parms[3];
```

```
if (height < 0.0)
    height =0.0;
// calculate the eased turbulence, taking into account the value
// may be greater than 1, which ease won't handle.
t_ease = height* inv_max_turb_length;
if(t_ease > 1.0)
  { t_ease = ((int)(t_ease)) +ease( (t_ease - ((int)t_ease)),
    .001, .999);
    if( t_ease > 2.5)
        t_ease = 2.5;
  }
else
  t_ease = ease(t_ease, .5, .999);
// Calculate the amount of turbulence to add in
fast_turb= fast_turbulence(pnt);
turb_amount = (fast_turb -0.875)* (.2 + .8*t_ease);
path_turb = fast_turb*(.2 + .8*t_ease);
// add turbulence to the height and see if it is above the top
height +=0.1*turb_amount;
if(height > end_d_ramp)
   { *density=0; return; }
//increase the radius of the column as the smoke rises
if(height <=0)
   rad_sq2 = rad_sq*.25;
else if (height <=end_d_ramp)
   { rad_sq2 = (.5 + .5*(ease( height/(1.75*end_d_ramp), .5,
               .5)))*radius;
     rad_sq2 *=rad_sq2;
   }
// ****************************************************
// move along a helical path
// ****************************************************

// calculate the path based on the unperturbed flow: helical path

hel_path.x = cos_theta2 *(1+ path_turb)*
           (1+cos(pnt_world.y*M_PI*2) *.11)
           *(1+ t_ease*.1) + big_radius*path_turb;
hel_path.z = sin_theta2 *(1+path_turb)*
           (1+sin(pnt_world.y*M_PI*2)*.085)*
           (1+ t_ease*.1) + .03*path_turb;
hel_path.y = - path_turb;
XYZ_ADD(direction2, pnt_world, hel_path);

//adjusting the center point for ramping off the density based on
//the turbulence of the moved point
turb_amount *= big_radius;
center.x = bottom.x - turb_amount;
center.z = bottom.z + .75*turb_amount;
//calculate the radial distance from the center and ramp off the
```

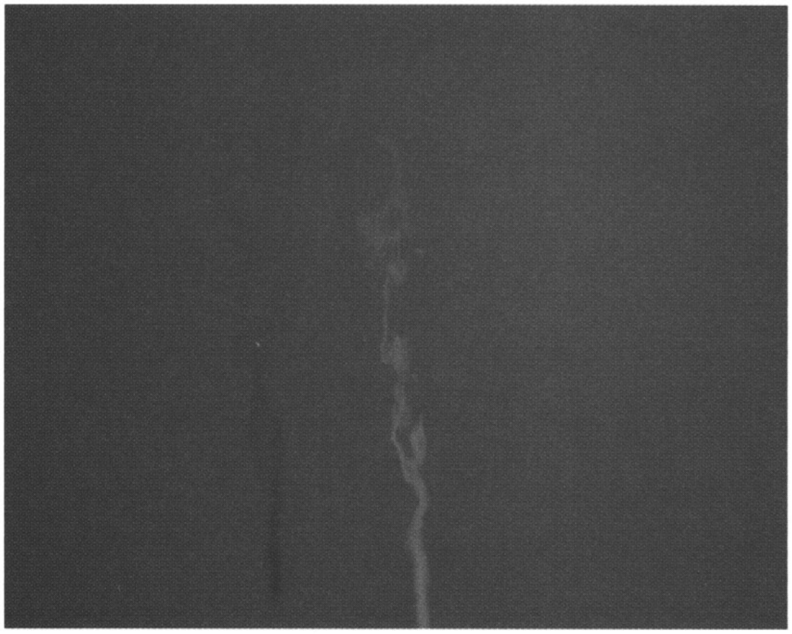

FIGURE 7.4 A rising column of smoke. Copyright © 1994 David S. Ebert.

```
// density based on this distance squared.
diff.x = center.x - direction2.x;
diff.z = center.z - direction2.z;
dist_sq = diff.x*diff.x + diff.z*diff.z;
if(dist_sq > rad_sq2)
  {*density=0; return;}
*density = (1-dist_sq/rad_sq2 + fast_turb*.05) *
           parms[1];
if(height > st_d_ramp)
   *density *= (1- ease((height - st_d_ramp)/(d_ramp_length), .5, .5));
}
```

The result of this procedure can be seen in Figure 7.4. In this procedure, turbulence is added to many variables before doing tests and computing paths. The addition of the turbulence produces a more natural appearance to the column of smoke. This procedure uses the same `offset` table as in the `steam_slab1` procedure. An `ease` procedure is also used to perform ease-in and ease-out of the turbulence addition and density ramping. The helical path that is used is actually the multiplication of two helical paths with the addition of turbulence. This calculation provides better

TABLE 7.2 PARAMETERS FOR SMOKE COLUMN PROCEDURE

`parm`	VALUE	DESCRIPTION
1	0.93	density scaling factor
2	1.6	height for 0 density (end of ramping it off)
3	0.175	height to start adding turbulence
4	0.685	height for maximum turbulence
5	0.0	height to start ramping density off
6	−0.88	`center.y`
7	2.0	speed for rising
8	0.04	radius
9	0.08	maximum radius of swirling

results than a single helical path. The parameter values and their description can be found in Table 7.2.

CONCLUSION

This chapter has provided an introduction to the solid space framework for procedural modeling and texturing and shown several example procedures for producing still images of gases. Animating these procedures is the topic of the next chapter, while Chapter 10 discusses methods for accelerating these techniques using graphics hardware.