8

ANIMATING SOLID SPACES

DAVID S. EBERT

The previous chapter discussed modeling the geometry of gases. This chapter discusses animating gases and other procedurally defined solid spaces. There are several ways that solid spaces can be animated. This chapter will consider two approaches:

1. Changing the solid space over time
2. Moving the point being rendered through the solid space

The first approach has time as a parameter that changes the definition of the space over time, a very natural and obvious way to animate procedural techniques. With this approach, time has to be considered in the design of the procedure, and the procedure evolves with time. Procedures that change the space to simulate growth, evolution, or aging are common examples of this approach. A related technique creates the procedure in a four-dimensional space, with time as the fourth dimension, such as a 4D noise function.

The second approach does not actually change the solid space, but moves the point in the volume or object over time through the space, in effect procedurally warping or perturbing the space. Moving the fixed three-dimensional screen space point along a path over time through the solid space before evaluating the turbulence function animates the gas (solid texture, hypertexture). Each three-dimensional screen space point is inversely mapped back to world space. From world space, it is mapped into the gas and turbulence space through the use of simple affine transformations. Finally, it is moved through the turbulence space over time to create the movement. Therefore, the path direction will have the reverse visual effect. For example, a *downward* path applied to the screen space point will show the texture or volume object *rising*. Figure 8.1 illustrates this process.

Both of these techniques can be applied to solid texturing, gases, and hypertextures. After a brief discussion of animation paths, the application of these two

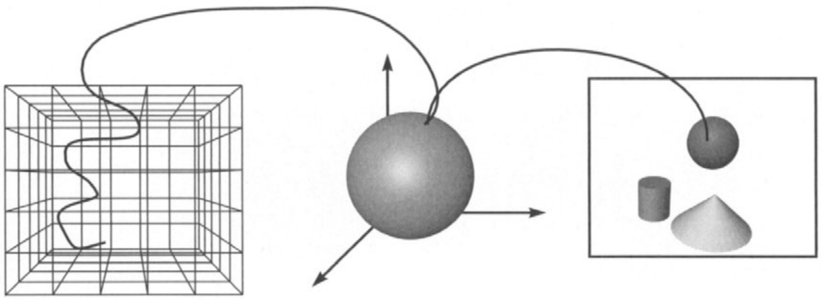

Figure 8.1 Moving a screen space point through the solid space.

techniques to solid texturing is discussed, followed by an exploration of the way they are used for gas animation and hypertextures, including liquids. Finally, the chapter concludes with a discussion of an additional procedural animation technique, particle systems.

ANIMATION PATHS

This chapter will describe various ways of creating animation paths for movement through the solid space. For many examples, I will use a helical (spiral) path. There are two reasons for using helical paths. First, most gases do not move along a linear path. Turbulence, convection, wind, and so on, change the path movement. From my observations, smoke, steam, and fog tend to swirl while moving in a given direction. A helical path can capture this general sense of motion. Second, helical paths are very simple to calculate. The calculation involves rotation around the axis of the helix (direction of motion) and movement along the axis. To create the rotation, the sine and cosine functions are used. The angle for these functions is based on the frame number to produce the rotation over time. The rate of rotation can be controlled by taking the frame number modulo a constant. Linear motion, again based on the frame number, will be used to create the movement along the axis.

The code segment below creates a helical path that rotates about the axis once every 100 frames. The speed of movement along the axis is controlled by the variable `linear_speed`.

```
theta  = (frame_number%100)*(2*M_PI/100);
path.x = cos(theta);
path.y = sin(theta);
path.z = theta*linear_speed;
```

One final point will clarify the procedures given in this chapter. To get smooth transitions between values and smooth acceleration and deceleration, *ease-in* and *ease-out* procedures are used. These are the standard routines used by animators to stop a moving object from jumping instantaneously from a speed of 0 to a constant velocity. One simple implementation of these functions assumes a sine curve for the acceleration and integrates this curve over one-half of its period.

ANIMATING SOLID TEXTURES

This section will show how the previous two animation approaches can be used for solid texturing. Applying these techniques to color solid texturing will be discussed first, followed by solid textured transparency.

A marble procedure will be used as an example of color solid texture animation. The following simple marble procedure is based on Perlin's `marble` function (Perlin 1985). An interactive hardware-accelerated version of `marble` is described in Chapter 10.

```
rgb_td marble(xyz_td pnt)
{
  float y;
  y = pnt.y + 3.0*turbulence(pnt, .0125);
  y = sin(y*M_PI);
  return (marble_color(y));
}
rgb_td marble_color(float x)
{
  rgb_td clr;
  x = sqrt(x+1.0)*.7071;
  clr.g = .30 + .8*x;
  x=sqrt(x);
  clr.r = .30 + .6*x;
  clr.b = .60 + .4*x;
  return (clr);
}
```

This procedure applies a sine function to the turbulence of the point. The resulting value is then mapped to the color. The results achievable by this procedure can be seen in Figure 8.2 (lower right).

Marble Forming

The application of the previous two animation approaches to this function has very different effects. When the first approach is used, changing the solid space over time,

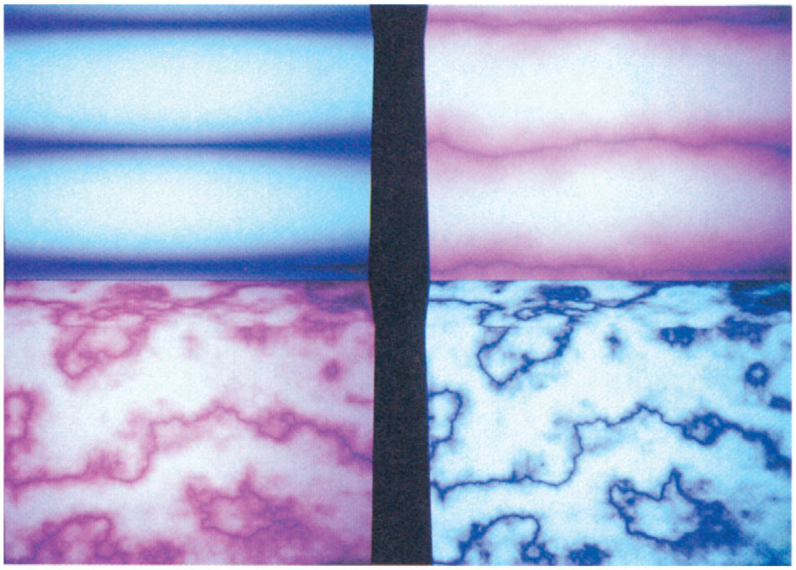

FIGURE 8.2 Marble forming. The images show the banded material heating, deforming, then cooling and solidifying. Copyright © 1992 David S. Ebert.

the formation of marble from banded rock can be achieved. Marble is formed from the turbulent mixing of different bands of rock. To simulate this process, initially no turbulence is added to the point; therefore, the sine function determines the color. Basing the color on the sine function produces banded material. As the frame number increases, the amount of turbulence added to the point is increased, deforming the bands into the marble vein pattern. The resulting procedure is the following:

```
rgb_td marble_forming(xyz_td pnt, int frame_num, int
                      start_frame, int end_frame)
 {
 float x, turb_percent, displacement;


 if(frame_num < start_frame)
    { turb_percent=0;
      displacement=0;
    }
   else if (frame_num >= end_frame)
    { turb_percent=1;
      displacement=3;
    }
   else
    { turb_percent= ((float)(frame_num-start_frame))/
```

```
                      (end_frame-start_frame);
      displacement = 3*turb_percent;
    }
x = pnt.x + turb_percent*3.0*turbulence(pnt, .0125) -
    displacement;
x = sin(x*M_PI);
return (marble_color(x));
}
```

The displacement value in this procedure is used to stop the entire texture from moving. Without the displacement value, the entire banded pattern moves horizontally to the left of the image, instead of the veins forming in place.

This procedure produces the desired effect, but the realism of the results can be increased by a few small changes. First of all, ease-in and ease-out of the rate of adding turbulence will give more natural motion. Second, the color of the marble can be changed to simulate heating before and during the deformation, and to simulate cooling after the deformation. The marble color is blended with a "glowing" marble color to simulate the heating and cooling. (Even though this may not be physically accurate, it produces a nice effect.) This can be achieved by the following procedure:

```
rgb_td marble_forming2(xyz_td pnt, int frame_num, int start_frame,
                      int end_frame, int heat_length)
 {
  float x, turb_percent, displacement, glow_percent;
  rgb_td m_color;
  if(frame_num < (start_frame-heat_length/2) 1½
     frame_num > end_frame+heat_length/2)
      glow_percent=0;
  else if (frame_num < start_frame + heat_length/2)
      glow_percent= 1.0 - ease( ((start_frame+heat_length/2-
                          frame_num)/ heat_length),0.4, 0.6);
  else if (frame_num > end_frame-heat_length/2)
      glow_percent = ease( ((frame_num-(end_frame-
                        heat_length/2))/heat_length),0.4, 0.6);
               else
      glow_percent=1.0;

  if(frame_num < start_frame)
    { turb_percent=0; displacement=0;
    }
  else if (frame_num >= end_frame)
    { turb_percent=1; displacement=3;
    }
  else
    { turb_percent= ((float)(frame_num-start_frame))/
                      (end_frame-start_frame);
      turb_percent=ease(turb_percent, 0.3, 0.7);
      displacement = 3*turb_percent;
    }
  x = pnt.y + turb_percent*3.0*turbulence(pnt, .0125) -
      displacement;
  x = sin(x*M_PI);
  m_color=marble_color(x);
  glow_percent= .5* glow_percent;
  m_color.r= glow_percent*(1.0)+ (1-glow_percent)*m_color.r;
  m_color.g= glow_percent*(0.4)+ (1-glow_percent)*m_color.g;
  m_color.b= glow_percent*(0.8)+ (1-glow_percent)*m_color.b;
  return(m_color);
 }
```

The resulting images can be seen in Figure 8.2. This figure shows four images of the change in the marble from banded rock (upper-left image) to the final marbled rock (lower-right image). Of course, the resulting sequence would be even more realistic if the material actually deformed, instead of the color simply changing. This effect will be described in the "Animating Hypertextures" section.

Marble Moving

A different effect can be achieved by the second animation approach, moving the point through the solid space. Any path can be used for movement through the marble space. A simple, obvious choice would be a linear path. Another choice, which produces very ethereal patterns in the material, is to use a turbulent path. The procedure below uses yet another choice for the path. This procedure moves the point along a horizontal helical path before evaluating the turbulence function, producing the effect of the marble pattern moving through the object. The helical path provides a more interesting result than the linear path, but does not change the general marble patterns as does using a turbulent path through the turbulence space. This technique can be used to determine the portion of marble from which to "cut" the object in order to achieve the most pleasing vein patterns. (You are in essence moving the object through a three-dimensional volume of marble.)

```
rgb_td moving_marble(xyz_td pnt, int frame_num)
 {
 float        x, tmp, tmp2;
 static float down, theta, sin_theta, cos_theta;
 xyz_td       hel_path, direction;
 static int   calcd=1;

 if(calcd)
    { theta=(frame_num%SWIRL_FRAMES)*SWIRL_AMOUNT;//swirling
      cos_theta = RAD1 * cos(theta) + 0.5;
      sin_theta = RAD2 * sin(theta) - 2.0;
      down = (float)frame_num*DOWN_AMOUNT+2.0;
      calcd=0;
    }
  tmp = fast_noise(pnt); // add some randomness
  tmp2 = tmp*1.75;
// calculate the helical path
  hel_path.y = cos_theta + tmp;
  hel_path.x = (-down) + tmp2;
  hel_path.z = sin_theta - tmp2;
  XYZ_ADD(direction, pnt, hel_path);
  x = pnt.y + 3.0*turbulence(direction, .0125);
  x = sin(x*M_PI);
  return (marble_color(x));
 }
```

In this procedure, SWIRL_FRAMES and SWIRL_AMOUNT determine the number of frames for one complete rotation of the helical path. By choosing SWIRL_FRAMES = 126 and SWIRL_AMOUNT = $2\pi/126$, the path swirls every 126 frames. DOWN_AMOUNT controls the speed of the downward movement along the helical path. A reasonable speed for downward movement for a unit-sized object is to use DOWN_AMOUNT = 0.0095. RAD1 and RAD2 are the y and z radii of the helical path.

Animating Solid Textured Transparency

This section describes the use of the second solid space animation technique, moving the point through the solid space, for animating solid textured transparency.

This animation technique is the one that I originally used for animating gases and is still the main technique that I use for gases. The results of this technique applied to solid textured transparency can be seen in Ebert, Boyer, and Roble (1989). The fog procedure given next is similar in its animation approach to the earlier moving_marble procedure. It produces fog moving through the surface of an object and can be used as a surface-based approach to simulate fog or clouds. Again in this procedure, a downward helical path is used for the movement through the space, which produces an upward swirling to the gas movement.

```
void fog(xyz_td pnt, float *transp, int frame_num)
{
 float tmp;
 xyz_td direction,cyl;
 double theta;
 pnt.x += 2.0 +turbulence(pnt, .1);
 tmp = noise_it(pnt);
 pnt.y += 4+tmp; pnt.z += -2 - tmp;
```

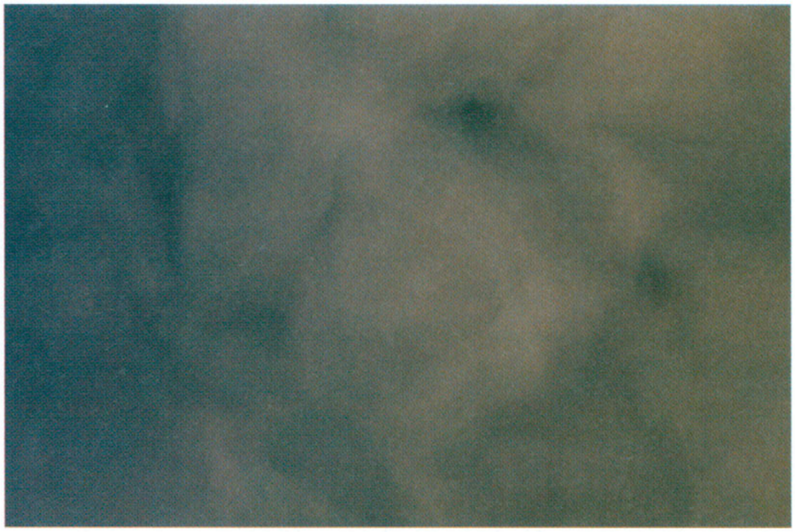

Figure 8.3 Solid textured transparency-based fog. Copyright © 1994 David S. Ebert.

```
theta =(frame_num%SWIRL_FRAMES)*SWIRL_AMOUNT;
 cyl.x =RAD1 * cos(theta); cyl.z =RAD2 * sin(theta);
 direction.x   = pnt.x + cyl.x;
 direction.y   = pnt.y - frame_num*DOWN_AMOUNT;
 direction.z   = pnt.z + cyl.z;
 *transp = turbulence(direction, .015);
 *transp = (1.0 -(*transp)*(*transp)*.275);
 *transp =(*transp)*(*transp)*(*transp);
}
```

An image showing this procedure applied to a cube can be seen in Figure 8.3. The values used for this image can be found in Table 8.1.

Another example of the use of solid textured transparency animation can be seen in Figure 8.4, which contains a still from an animation entitled *Once a Pawn a Foggy Knight*... (Ebert, Boyer, and Roble 1989). In this scene, three planes are positioned to give a two-dimensional approximation of three-dimensional fog. One plane is in front of the scene, one plane is approximately in the middle, and the final plane is behind all the objects in the scene.

This technique is similar to Gardner's technique for producing images of clouds (Gardner 1985), except that it uses turbulence to control the transparency instead of Fourier synthesis. As with any surface-based approach to modeling gases, including

TABLE 8.1 VALUES FOR FOG PROCEDURE

PARAMETER	VALUE
DOWN_AMOUNT	0.0095
SWIRL_FRAMES	126
SWIRL_AMOUNT	$2\pi/126$
RAD1	0.12
RAD2	0.08

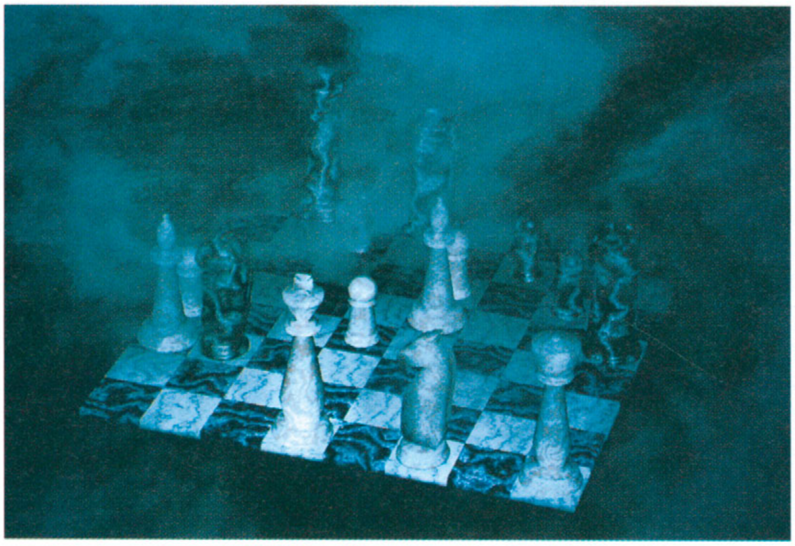
FIGURE 8.4 A scene from *Once a Pawn a Foggy Knight . . .* showing solid textured transparency used to simulate fog. Copyright © 1989 David S. Ebert.

Gardner's, this technique cannot produce three-dimensional volumes of fog or accurate shadowing from the fog.

ANIMATION OF GASEOUS VOLUMES

As described in the previous section, animation technique 2, moving the point through the solid space, is the technique that I use to animate gases. This technique will be used in all the examples in this section. Moving each fixed three-dimensional screen space point along a path over time through the solid space before evaluating the turbulence function creates the gas movement. First, each three-dimensional screen space point is inversely mapped back to world space. Second, it is mapped from world space into the gas and turbulence space through the use of simple affine transformations. Finally, it is moved through the turbulence space over time to create the movement of the gas. Therefore, the path direction will have the reverse visual effect. For example, a downward path applied to the screen space point will cause the gas to rise.

This gas animation technique can be considered to be the inverse of particle systems because each point in three-dimensional screen space is moved through the gas space to see which portion of the gas occupies the current location in screen space. The main advantage of this approach over particle systems is that extremely large geometric databases of particles are not required to get realistic images. The complexity is always controlled by the number of screen space points in which the gas is potentially visible.

Several interesting animation effects can be achieved through the use of helical paths for movement through the solid space. These helical path effects will be described first, followed by the use of three-dimensional tables for controlling the gas movement. Finally, several additional primitives for creating gas animation will be presented.

Helical Path Effects

Helical paths can be used to create several different animation effects for gases. In this chapter, three examples of helical path effects will be presented: steam rising from a teacup, rolling fog, and a rising column of smoke.

Steam Rising from a Teacup

In the previous chapter, a procedure for producing a still image of steam rising from a teacup was described. This procedure can be modified to produce convincing animations of steam rising from the teacup by the addition of helical paths for motion. Each point in the volume is moved downward along a helical path to produce the steam rising and swirling in the opposite direction. The modification needed is given below. This animation technique is the same technique that was used in the `moving_marble` procedure.

```
    void steam_moving(xyz_td pnt, xyz_td pnt_world, float *density,
                      float *parms, vol_td vol)
    {
*** float noise_amt,turb, dist_sq, density_max, offset2, theta, dist;
    static float pow_table[POW_TABLE_SIZE], ramp[RAMP_SIZE],
                 offset[OFFSET_SIZE];
    extern int frame_num;
    xyz_td direction, diff;
    int i, indx;
    static int calcd=1;
*** static float down, cos_theta, sin_theta;

    if(calcd)
      { calcd=0;
       // determine how to move point through space(helical path)
***    theta =(frame_num%SWIRL_FRAMES)*SWIRL;
***    down = (float)frame_num*DOWN*3.0 +4.0;
***    cos_theta = RAD1*cos(theta) +2.0;
***    sin_theta = RAD2*sin(theta) -2.0;
       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);
       }
    // move the point along the helical path
*** noise_amt = fast_noise(pnt);
*** direction.x = pnt.x + cos_theta + noise_amt;
*** direction.y = pnt.y - down + noise_amt;
*** direction.z = pnt.z +sin_theta + noise_amt;
    turb =fast_turbulence(direction);
    *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;
      }
    // ramp it off vertically
    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)
        { offset2 = (dist -.05)*1.111111;
         offset2 = 1 - (exp(offset2)-1.0)71.718282;
         offset2*=parms[1];
         *density *= offset2;
        }
   }
  }
```

The lines that have changed from the earlier steam_slab1 procedure are marked with three asterisks (***). This procedure creates upward swirling movement in the gas, which swirls around 360 degrees every SWIRL_FRAMES frame. Noise is applied to the path to make it appear more random. The parameters RAD1 and RAD2 determine the elliptical shape of the swirling path. Additional variables in this procedure are the angle of rotation about the helical path (theta), the frame number (frame_num), the cosine of the angle of rotation (cos_theta), the sine of the angle of rotation (sin_theta), the amount to move along the helical axis (down), a noise amount to add to the path (noise_amt), and the new location of the point after movement along the path (direction).

The downward helical path through the gas space produces the effect of the gas rising and swirling in the opposite direction.

For more realistic steam motion, a simulation of air currents is helpful. Adding turbulence to the helical path can approximate this, where the amount of turbulence added is proportional to the height above the teacup. (This assumes that no turbulence is added at the surface.)

Fog Animation

The next example of helical path effects is the creation of rolling fog. For this animation, a horizontal helical path will be used to create the swirling motion of the fog to the right of the scene. From examining the following volume_fog_animation procedure, it is clear that this procedure uses the same animation technique as the earlier steam_moving procedure: move each point along a helical path before evaluating the turbulence function. The value returned by the turbulence function is again multiplied by a density scalar factor, parms[1], and raised to a power, parms[2]. As in the previous procedures, a precomputed table of density values raised to a power is used to speed calculation. A more complete description of the use of helical paths for producing fog animation can be found in Ebert and Parent (1990).

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

```
{
 float noise_amt, turb;
 extern int frame_num;
 xyz_td direction;
 int indx;
 static float pow_table[POW_TABLE_SIZE];
 int i;
 static int calcd=1;
 static float down, cos_theta, sin_theta, theta;
 if(calcd)
   {
    down = (float)frame_num*SPEED*1.5 +2.0;
    theta =(frame_num%SWIRL_FRAMES)*SWIRL_AMOUNT;//get swirling effect
    cos_theta = cos(theta)*.1 + 0.5; //use a radius of .1
    sin_theta = sin(theta)*.14 - 2.0; //use a radius of .14
    calcd=0;
    for(i=POW_TABLE_SIZE-1; i>=0; i--)
     {
      pow_table[i]=(float)pow(((double)(i))/(POW_TABLE_SIZE-1)*
               parms[1]*4.0,(double)parms[2]);
     }
   }
 // make it move horizontally & add some noise to the movement
 noise_amt = fast_noise(pnt);
 direction.x = pnt.x - down + noise_amt*1.5;
 direction.y = pnt.y + cos_theta +noise_amt;
 direction.z = pnt.z + sin_theta -noise_amt*1.5;
 // base the turbulence on the new point
 turb =fast_turbulence(direction);
 *density = pow_table[(int)((turb*turb)*(.25*(POW_TABLE_SIZE-1)))];
 // make sure density isn't greater than 1
 if(*density >1)
   *density=1;
}
```

As in the fog and steam_moving procedures, the volume_fog_animation procedure uses the same values for SWIRL_FRAMES (126) and SWIRL_AMOUNT ($2\pi/126$). SPEED controls the rate of horizontal movement, and the value I use to produce gently rolling fog is 0.012. The results achievable by this procedure can be seen in Figure 8.5, which is a still from an animation entitled *Getting into Art* (Ebert, Ebert, and Boyer 1990). For this image, parms[1] = 0.22 and parms[2] = 4.0.

Smoke Rising

The final example of helical path effects is the animation of the smoke_stream procedure given earlier to create a single column of smoke. Two different helical paths are

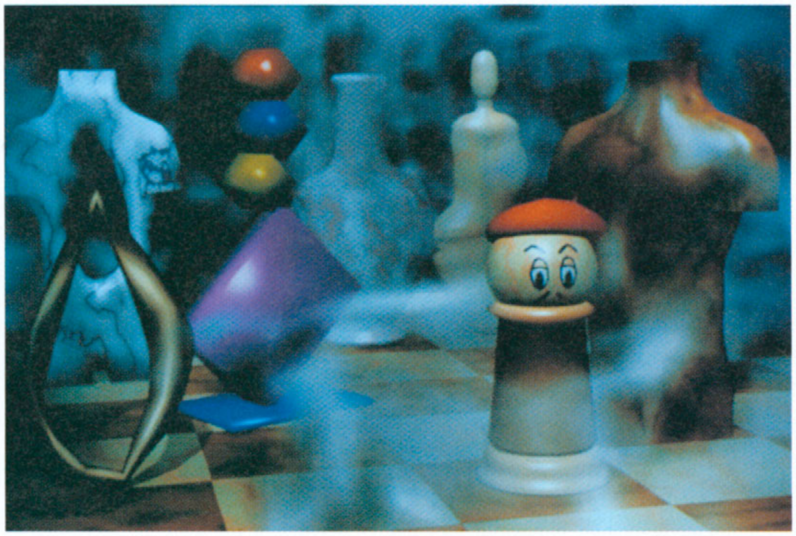

FIGURE 8.5 A scene from *Getting into Art,* showing volume-rendered fog animation created by horizontal helical paths. Copyright © 1990 David S. Ebert.

used to produce the swirling column of smoke. This `smoke_stream` procedure already used a helical path to displace each point to get a more convincing column of smoke. We will now modify this helical path to make it a downward helical path based on the frame number, creating the rising column of smoke. The second helical path will actually displace the center point of the cylinder, producing a swirling cylinder of smoke (instead of a vertical cylinder as was used in Chapter 7). This second helical path will swirl at a different rate than the first. The same input parameter values can be used for this procedure. The following is the procedure that is the result of these modifications.

```
// ********************************************************
//  Rising_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 off density
// parms[6] = center.y
// parms[7] = speed for rising
// parms[8] = radius
// parms[9] = max radius of swirling
// *******************************************************
void rising_smoke_stream(xyz_td pnt,float *density, float
                        *parms, xyz_td pnt_world, vol_td *vol)
{
 float dist_sq;
 extern float offset[OFFSET_SIZE];
 extern int frame_num;
 static int calcd=1;
 static float down, cos_theta2, sin_theta2;
 xyz_td hel_path, center, diff, direction2;
 double ease(), turb_amount, theta_swirl, cos_theta, sin_theta;
 static xyz_td bottom;
 static double rad_sq, max_turb_length, radius, big_radius,
              st_d_ramp, d_ramp_length, end_d_ramp, down3,
              inv_max_turb_length, cos_theta3, sin_theta3;
 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];
    st_d_ramp =MIN(st_d_ramp, end_d_ramp);
    end_d_ramp = parms[2];
    d_ramp_length = end_d_ramp - st_d_ramp;
    //calculate rotation about the helix axis based on frame_number

*** theta_swirl=(frame_num%SWIRL_FRAMES_SMOKE)*SWIRL_SMOKE; // swirling
*** cos_theta = cos(theta_swirl);
*** sin_theta = sin(theta_swirl);
*** down = (float)(frame_num)*DOWN_SMOKE*.75 * parms[7];
    // Calculate sine and cosine of the different radii of the
    // two helical paths
*** cos_theta2 = .01*cos_theta;
*** sin_theta2 = .0075*sin_theta;
*** cos_theta3= cos_theta2*2.25;
*** sin_theta3= sin_theta2*4.5;
*** down3= down*2.25;
    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);
  // move point along the helical path before evaluating turbulence
*** pnt.x += cos_theta3;
*** pnt.y -= down3;
*** pnt.z += sin_theta3;
    fast_turb= fast_turbulence_three(pnt);
    turb_amount = (fast_turb -0.875)* (.2 + .8*t_ease);
    path_turb = fast_turb*(.2 + .8*t_ease);
    // add turbulence to the height & 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;
      }
    else
       rad_sq2 = rad_sq;
    //
    // move along a helical path plus add the ability to use tables
    //
    // calculate the path based on the unperturbed flow: helical path
    //
*** hel_path.x = cos_theta2 *(1+path_turb)*(1+t_ease*.1)*
       (1+cos((pnt_world.y+down*.5)*M_PI*2)*.11) + big_radius*path_turb;
*** hel_path.z = sin_theta2 * (1+path_turb)*(1+ t_ease*.1)*
       (1+sin((pnt_world.y +down*.5)*M_PI*2)*.085) + .03*path_turb;
*** hel_path.y = (- down) - path_turb;
    XYZ_ADD(direction2, pnt_world, hel_path);
    // adjusting center point for ramping off 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 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 statements that have been changed from the smoke_stream procedure are marked with three asterisks (***). As can be seen, the main changes are in calculating and using two helical paths based on the frame number. One path displaces the center of the cylinder, and the point being rendered is moved along the other path. After trials with only one helical path, it becomes clear that two helical paths give a better effect. Figure 8.6 shows the results of this rising_smoke_stream procedure. This figure contains three images from an animation of rising smoke.

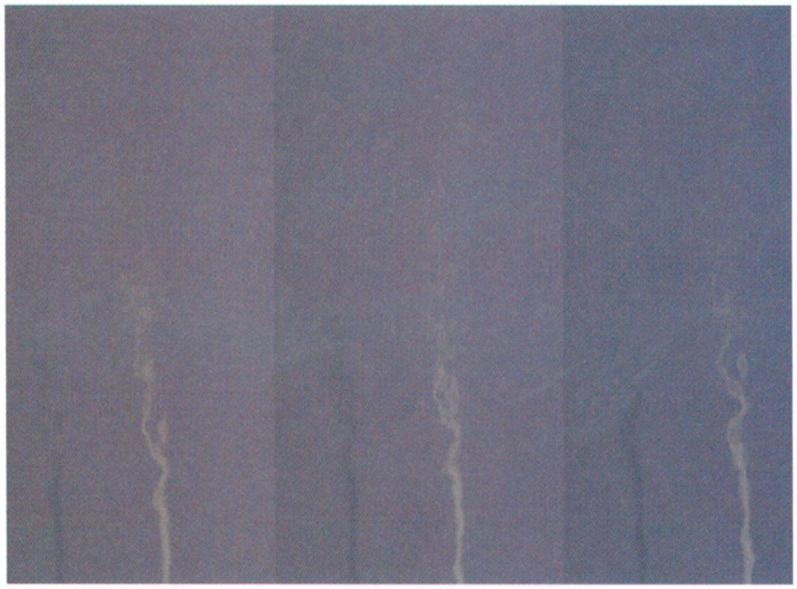

FIGURE 8.6 Rising column of smoke animation. Images are every 30 frames. Copyright © 1994 David S. Ebert.

THREE-DIMENSIONAL TABLES

As shown above, a wide variety of effects can be achieved through the use of helical paths. These aforementioned procedures require the same type of path to be used for movement throughout the entire volume of gas. Obviously, more complex motion can be achieved by having different path motions for different locations within the gas. A three-dimensional table specifying different procedures for different locations within the volume is a good, flexible solution for creating complex motion in this manner.

The use of three-dimensional tables (solid spaces) to control the animation of the gases is an extension of my previous use of solid spaces in which three-dimensional tables were used for volume shadowing effects (Ebert and Parent 1990).

The three-dimensional tables are handled in the following manner: The table surrounds the gas volume in world space, and values are stored at each of the lattice points in the table (see Figure 8.7). These values represent the calculated values for that specific location in the volume. To determine the values for other locations in the volume, the eight table entries forming the parallelepiped surrounding the point are interpolated. For speed in accessing the table values, I currently require table dimensions to be powers of two and actually store the three-dimensional table as a one-dimensional array. This restriction allows the use of simple bit-shifting operations in determining the array index. These tables could be extended to have nonuniform spacing between table entries within each dimension, creating an octree-like

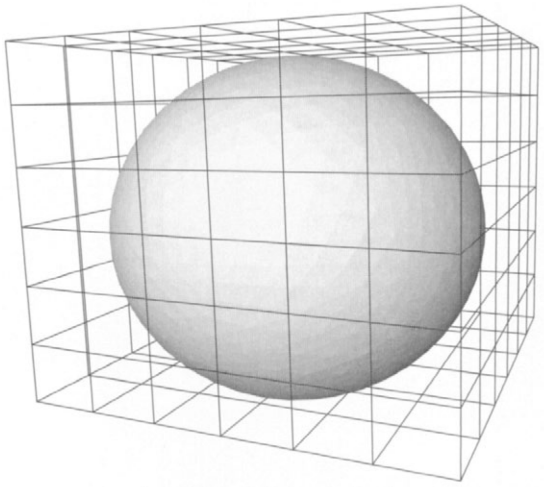
Figure 8.7

structure; however, this would greatly increase the time necessary to access values from the table, as this fast bit-shifting approach could no longer be used. Table dimensions are commonly of the order of 64 × 64 × 64 or 128 × 64 × 32.

I use two types of tables for controlling the motion of the gases: vector field tables and functional flow field tables. The vector field tables store direction vectors, density scaling factors, and other information for their use at each point in the lattice. Therefore, these tables are suited for visualizing computational fluid dynamics simulations or using external programs for controlling the gas motion. The vector field tables will not be described in this chapter. A thorough description of their use and merits can be found in Ebert (1991). This chapter concentrates on the use of functional flow field tables for animation control.

The functional flow field and vector field tables are incorporated into the volume density functions for controlling the shape and movement of the gas. Each volume density function has a default path and velocity for the gas movement. First, the default path and velocity are calculated; second, the vector field tables are evaluated; and, finally, functions that calculate direction vectors, density scaling factors, and so on, from the functional flow field tables are applied. The default path vector, the vector from the vector field table, and the vector from the flow field function are combined to produce the new path for the movement through the gas space.

Accessing the Table Entries

When values are accessed from these tables during rendering, the location of the sample point within the table is determined. As mentioned earlier, this point will lie within a parallelepiped formed by the eight table entries that surround the point. The values at these eight points are interpolated to determine the final value. The location within the table is determined by first mapping the three-dimensional screen space point back into world space. The following formula is then used to find the location of the point within the table:

```
ptable.x = (point.x-table_start.x) * table_inv_step.x
ptable.y = (point.y-table_start.y) * table_inv_step.y
ptable.z = (point.z-table_start.z) * table_inv_step.z
```

`Ptable` is the location of the point within the three-dimensional table, which is determined from `point`, the location of the point in world space. `table_start` is the location in world space of the starting table entry, and `table_inv_step` is the inverse of the step size between table elements in each dimension. Once the location within the table is determined, the values corresponding to the eight surrounding table entries are then interpolated (trilinear interpolation should suffice).

Functional Flow Field Tables

Functional flow field tables are a valuable tool for choreographing gas animation. These tables define, for each region of the gas, which functions to evaluate to control the gas movement. Each flow field table entry can contain either one specific function to evaluate or a list of functions to evaluate to determine the path for the gas motion (path through the gas space). For each function, a file is specified that contains the type of function and parameters for that function. The functions evaluated by the flow field tables return the following information:

- Direction vector
- Density scaling value
- Percentage of vector to use
- Velocity

The advantage of using flow field functions is that they can provide infinite detail in the motion of the gas. They are not stored at a fixed resolution, but are evaluated for each point that is volume rendered. The disadvantage is that the functions are much more expensive to evaluate than simply interpolating values from the vector field table.

The "percentage of vector to use" value in the previous list is used to provide a smooth transition between control of the gas movement by the flow field functions, the vector field tables, and the default path of the gas. This value is also used to allow a smooth transition between control of the gas by different flow field functions. This value will decrease as the distance from the center of control of a given flow field function increases.

Functional Flow Field Functions

Two powerful types of functions for controlling the movement of the gases are attractors/repulsors and vortex functions. Repulsors are the *exact* opposite of attractors, so only attractors will be described here. To create a repulsor from an attractor, simply negate the direction vector.

All of the following procedures will take as input the location of the point in the solid space (`pnt`) and a structure containing parameters for each instance of the function (`ff`). These procedures will return a density scaling factor (`density_scaling`), the direction vector for movement through the gas space (`direction`), the percentage of this vector to use in determining the motion through the gas space (`percent_to_use`), and a velocity scaling factor (`velocity`). The `density_scaling` parameter allows these procedures to decrease or increase the gas density as it moves through a region of space. The `velocity` parameter similarly allows these procedures to change the velocity of the gas as it moves through a region of space. The most important parameters, however, are the `direction` and `percent_to_use` parameters, which are used to determine the path motion through the solid space.

Attractors

Attractors are primitive functions that can provide a wide range of effects. Figure 8.8 shows several frames of an attractor whose attraction increases in strength over time. Each attractor has a minimum and maximum attraction value. In this figure, the interpolation varies over time between the minimum and maximum attraction values of the attractor. By animating the location and strength of an attractor, many different effects can be achieved. Effects such as a breeze blowing (see Figure 8.9) and the wake of a moving object are easy to create. Spherical attractors create paths radially away from the center of attraction (as stated previously, path movement needs to be in the opposite direction of the desired visual effect). The following is an example of a simple spherical attractor function:

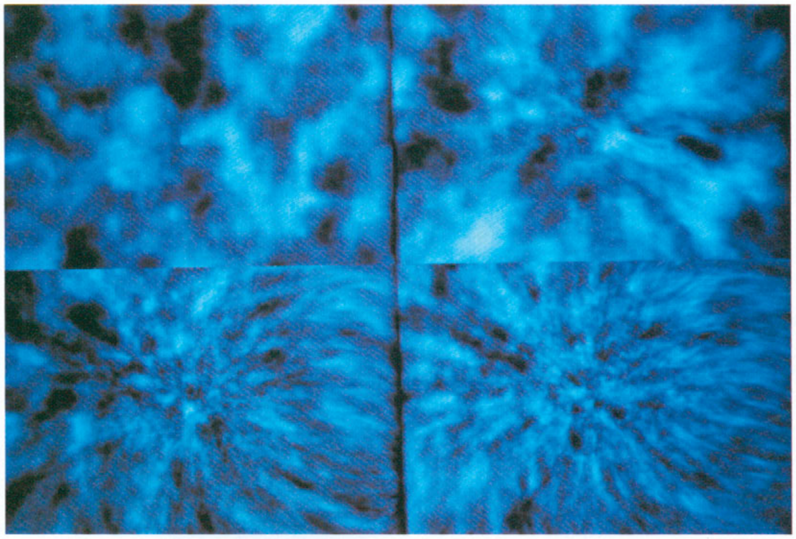
Figure 8.8 Effect of a spherical attractor increasing over time. Images are every 45 frames. The top-left image has 0 attraction. The lower-right image has the maximum attraction. Copyright © 1992 David S. Ebert.

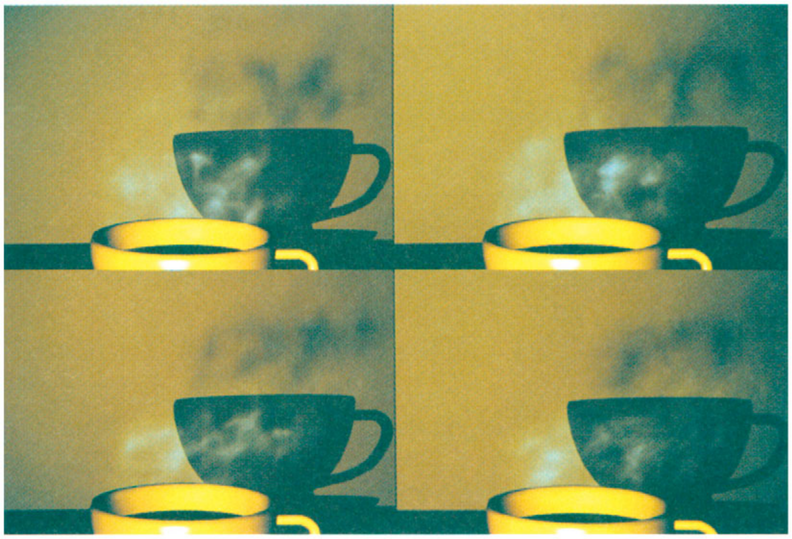
FIGURE 8.9 An increasing breeze blowing toward the right created by an attractor. Copyright © 1991 David S. Ebert.

```
void spherical_attractor(xyz_td point, flow_func_td ff, xyz_td
                         *direction, float *density_scaling,
                         float *velocity, float *percent_to_use)
 {
 float dist, d2;
 // calculate distance & direction from center of attractor
 XYZ_SUB(*direction, point, ff.center);
 dist=sqrt(DOT_XYZ(*direction,*direction));
 // set the density scaling and the velocity to 1
 *density_scaling=1.0;
 *velocity=1.0;
 // calculate the falloff factor (cosine)
 if(dist > ff.distance)
    *percent_to_use=0;
 else if (dist < ff.falloff_start)
    *percent_to_use=1.0;
 else
   { d2 =(dist-ff.falloff_start)/(ff.distance-
                                  ff.falloff_start);
 *percent_to_use = (cos(d2*M_PI)+1.0)*.5;
   }
}
```

The `flow_func_td` structure contains parameters for each instance of the spherical attractor. The parameters include the center of the attractor (`ff.center`), the effective distance of attraction (`ff.distance`), and the location to begin the fall-off from the attractor path to the default path (`ff.falloff_start`). This function ramps the use of the attractor path from `ff.falloff_start` to `ff.distance`. A cosine function is used for a smooth transition between the path defined by the attractor and the default path of the gas.

Extensions of Spherical Attractors

Variations on this simple spherical attractor include moving attractors, angle-limited attractors, attractors with variable maximum attraction, nonspherical attractors, and, of course, combinations of any or all of these types.

One variation on the preceding spherical attractor procedure is to animate the location of the center of attraction. This allows for dynamic animation control of the gas. Another useful variation is angle-limited attractors. As opposed to having the range of the attraction being 360 degrees, an axis and an angle for the range of attraction can be specified. This can be implemented in a manner very similar to angle-limited light sources and can be animated over time. These two variations can be combined to produce interesting effects. For example, an angle-limited attractor following the movement of the object can create a wake from a moving object. This attractor will cause the gas behind the object to be displaced and pulled in the direction of the moving object. The minimum and maximum attraction of the attractor can also be animated over time to produce nice effects as seen in Figures 8.8 and 8.9. Figure 8.8 shows an attractor increasing in strength over time, and Figure 8.9 shows a breeze blowing the steam rising from a teacup. As will be described later, the breeze is simulated with an animated attractor.

The geometry of the attraction can be not only spherical, but also planar or linear. A linear attractor can be used for creating the flow of a gas along a wall, as will be explained later.

Spiral Vortex Functions

Vortex functions have a variety of uses, from simulating actual physical vortices to creating interesting disturbances in flow patterns as an approximation of turbulent flow. The procedures described are not attempts at a physical simulation of vortices—an extremely complex procedure requiring large amounts of supercomputer time for approximation models.

One vortex function is based on the simple 2D polar coordinate function

$$r = \theta$$

which translates into three-dimensional coordinates as

$$x = \theta \times \cos(\theta)$$
$$y = \theta \times \sin(\theta)$$

The third dimension is normally linear movement over time along the third axis. To animate this function, θ is based on the frame number. To increase the vortex action, a scalar multiplier for the sine and cosine terms based on the distance from the vortex's axis is added. This polar equation alone produces swirling motion; however, more convincing vortices can be created by the modifications described below, which base the angle of rotation on both the frame number and the distance from the center of the vortex. The resulting vortex procedure is the following:

```
void calc_vortex(xyz_td *pt, flow_func_td *ff, xyz_td *direction,
    float *velocity, float *percent_to_use, int frame_num)
 {
 static tran_mat_td mat={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
 xyz_td      dir, pt2, diff;
 float       theta, dist, d2, dist2;
 float       cos_theta, sin_theta, compl_cos, ratio_mult;
 // calculate distance from center of vortex
XYZ_SUB(diff,(*pt), ff->center);
dist=sqrt(DOT_XYZ(diff,diff));
 dist2 = dist/ff->distance;
 // calculate angle of rotation about the axis
 theta = (ff->parms[0]*(1+.001*(frame_num)))/
        (pow((.1+dist2*.9), ff->parms[1]));
  // calculate matrix for rotating about the cylinder's axis
 calc_rot_mat(theta, ff->axis, mat);
transform_XYZ((long)1,mat,pt,&pt2);
 XYZ_SUB(dir,pt2,(*pt));
 direction->x = dir.x;
 direction->y = dir.y;
 direction->z = dir.z;
  // Have the maximum strength increase from frame parms[4] to
  // parms[5] to a maximum of parms[2]
 if(frame_num < ff->parms[4])
  ratio_mult=0;
 else if (frame_num <= ff->parms[5])
  ratio_mult = (frame_num - ff->parms[4])/
          (ff->parms[5] - ff->parms[4])* ff->parms[2];
else
 ratio_mult = ff->parms[2];
 //calculate the falloff factor
if(dist > ff->distance)
  { *percent_to_use=0;
   *velocity=1;
  }
else if (dist < ff->falloff_start)
  { *percent_to_use=1.0 *ratio_mult;
   // calc velocity
   *velocity= 1.0+(1.0 - (dist/ff->falloff_start));
  }
else
  { d2 =(dist-ff->falloff_start)/(ff->distance -
                                 ff->falloff_start);
   *percent_to_use = (cos(d2*M_PI)+1.0)*.5*ratio_mult;
   *velocity= 1.0+(1.0 - (dist/ff->falloff_start));
   }
}
```

This procedure uses the earlier polar function in combination with suggestions from Karl Sims (1990) to produce the vortex motion. For these vortices, both the frame number and the relative distance of the point from the center (or axis) of rotation determine the angle of rotation about the axis. The direction vector is then the vector difference of the transformed point and the original point. The `calc_vortex` procedure also allows the animation of the strength of the vortex action.

A third type of vortex function is based on the conservation of angular momentum: $r * q = constant$, where r is the distance from the center of the vortex. This formula can be used in the earlier vortex procedure to calculate the angle of rotation about the axis of the vortex: $\theta = (time * constant)/r$. The angular momentum will be conserved, producing more realistic motion.

An example of the effects achievable by the previous vortex procedure can be seen in Figure 8.10. Animating the location of these vortices produces interesting effects, especially when coordinating their movement with the movement of objects in the scene, such as a swirling wake created by an object moving through the gas.

Combinations of Functions

The real power of flow field functions is the ability to combine these primitive functions to control the gas movement through different volumes of space. The combination of flow field functions provides very interesting and complex gas motion. Two examples of the combination of flow field functions, wind blowing and flow into a hole, are presented next to illustrate the power of this technique.

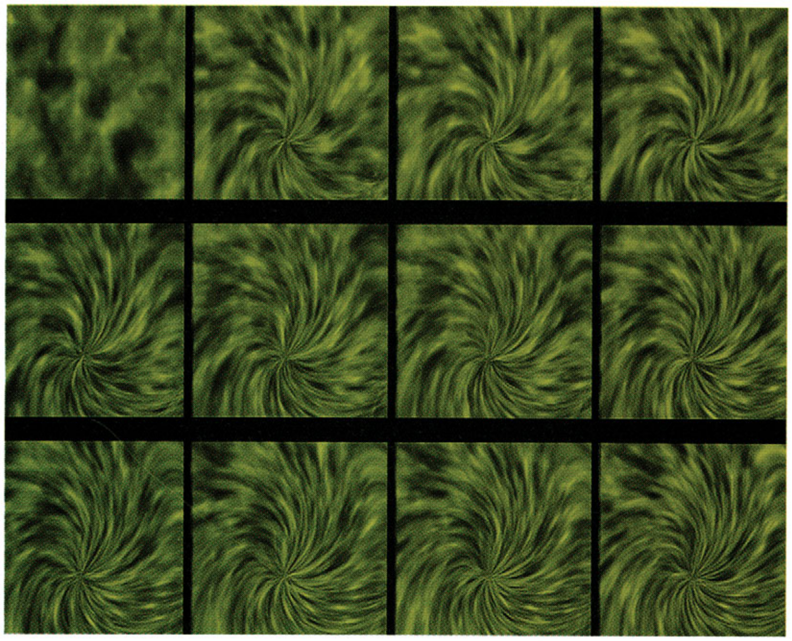

FIGURE 8.10 Spiral vortex. Images are every 21 frames. The top-left image is the default motion of the gas. The remaining images show the effects of the spiral vortex. Copyright © 1994 David S. Ebert.

Wind Effects

The first complex gas motion example is wind blowing the steam rising from a teacup. A spherical attractor is used to create the wind effect. Figure 8.9 shows frames of an animation of a breeze blowing the steam from the left of the image. To produce this effect, an attractor was placed to the upper right of the teacup and the strength of attraction was increased over time. The maximum attraction was only 30%, producing a light breeze. An increase in the maximum attraction would simulate an increase in the strength of the wind. The top-left image shows the steam rising vertically with no effect of the wind. The sequence of images (top-right image to bottom-right image) shows the effect on the steam as the breeze starts blowing toward the right of the image. This is a simple combination of helical motion with an attractor. Notice how the volume of the steam, as well as the motion of the individual plumes, is "blown" toward the upper right. This effect was created by moving the center of the volume point for the ramping of the density over time. The x-value of the center point is increased, based on the height from the cup and the frame number. By changing the spherical_attractor flow function and the steam_moving procedure given earlier, the blowing effect can be implemented. The following is the addition needed to the spherical_attractor procedure:

```
// **************************************************
// Move the Volume of the Steam
// Shifting is based on the height above the cup
// (parms[6]->parms[7]) and the frame range for increasing
// the strength of the attractor. This is from ratio_mult
// that is calculated above in calc_vortex.
// **************************************************
// Have the maximum strength increase from frame parms[4] to
// parms[5] to a maximum of parms[2]
if(frame_num < ff->parms[4])
  ratio_mult=0;
else if (frame_num <= ff->parms[5])
  ratio_mult = (frame_num - ff->parms[4])/
   (ff->parms[5] - ff->parms[4]) * ff->parms[2];
if(point.y < ff->parms[6])
   x_disp=0;
 else
 {if(point.y <= ff->parms[7])
   d2=COS_ERP((point.y-ff->parms[6])/
              (ff->parms[7]-ff->parms[6]));
  else
   d2=0;
   x_disp=(1-d2)*ratio_mult*parms[8]+fast_noise(point)*
          ff->parms[9];
}
return(x_disp);
```

Table 8.2 clarifies the use of all the parameters. The ratio_mult value for increasing the strength of the attraction is calculated in the same way as in the calc_vortex procedure. The x_disp value needs to be returned to the steam_rising function. This value is then added to the center variable before the density is ramped off. The following addition to the steam_rising procedure will accomplish this:

```
center = vol.shape.center;
center.x += x_disp;
```

Flow into a Hole in a Wall

The next example of combining flow field functions constrains the flow into an opening in a wall. The resulting images are shown in Figure 8.11. Figure 8.11(a)

TABLE 8.2 PARAMETERS FOR WIND EFFECTS

VARIABLE	DESCRIPTION
`point`	location of the point in world space
`ff → parms[2]`	maximum strength of attraction
`ff → parms[4]`	starting frame for attraction increasing
`ff → parms[5]`	ending strength for attraction increasing
`ff → parms[6]`	minimum *y*-value for steam displacement
`ff → parms[7]`	maximum *y*-value for steam displacement
`ff → parms[8]`	maximum amount of steam displacement
`ff → parms[9]`	amount of noise to add in

shows gas flowing into an opening in a wall on the right of the image. Figure 8.11(b) shows liquid flowing into the opening. For this example, three types of functions are used. The first function is an angle-limited spherical attractor placed at the center of the hole. This attractor has a range of 180 degrees from the axis of the hole toward the left. The next function is an angle-limited repulsor placed at the same location, again with a range of repulsion of 180 degrees, but to the right of the hole. These two functions create the flow into the hole and through the hole. The final type of function creates the tangential flow along the walls. This function can be considered a linear attraction field on the left side of the hole. The line in this case would be through the hole and perpendicular to the wall (horizontal). This attractor has maximum attraction near the wall, with the attraction decreasing as the distance from the wall increases. As can be seen from the flow patterns toward the hole and along the wall in Figure 8.11, the effect is very convincing. This figure also shows how these techniques can be applied to hypertextures. Figure 8.11(b) is rendered as a hypertexture to simulate a (compressible) liquid flowing into the opening.

ANIMATING HYPERTEXTURES

All of the animation techniques described above can be applied to hypertextures; only the rendering algorithm needs to be changed. The volume density functions that I use for gases are, in reality, hypertexture functions. The difference is that an atmospheric rendering model is used. Therefore, by using a nongaseous model for illumination and for converting densities to opacities, the techniques described above will

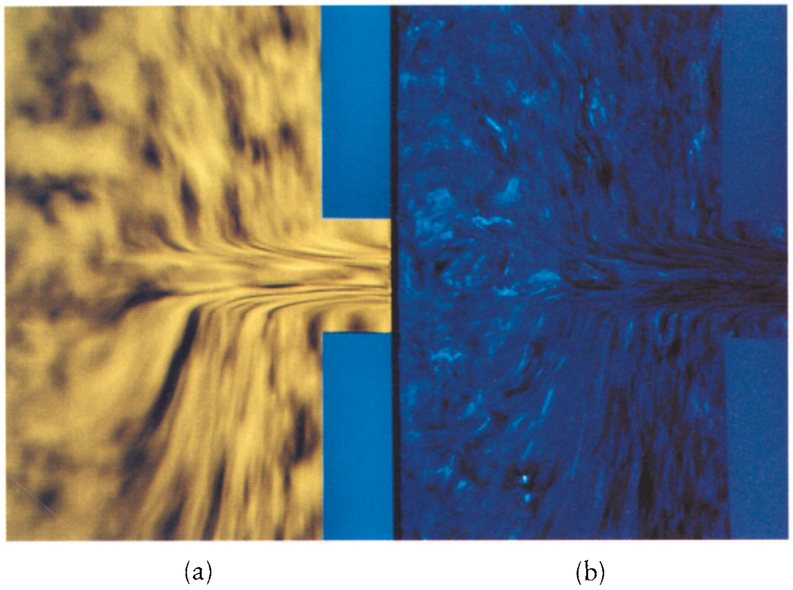

FIGURE 8.11 (a) Gas flowing into a hole in the wall. (b) Liquid flowing into a hole in the wall. Copyright © 1991 David S. Ebert.

produce hypertexture animations. An example of this is Figure 8.11(b). The geometry and motion procedures are the same for both of the images in Figure 8.11.

Volumetric Marble Formation

One other example of hypertexture animation will be explored: simulating marble formation. The addition of hypertexture animation to the solid texture animation discussed earlier will increase the realism of the animation considerably.

One approach is to base the density changes on the color of the marble. Initially, no turbulence will be added to the "fluid": density values will be determined in a manner similar to the marble color values, giving the different bands different densities. Just as in the earlier `marble_forming` procedure, turbulence will be added over time. In the following procedure, these changes are achieved by returning the amount of turbulence, `turb_amount`, from the solid texture function, `marble_forming`, described earlier. The density is based on the turbulence amount from the solid texture function. This is then shaped using the power function in a similar manner to the gas functions given before. Finally, a trick by Perlin (subtracting 0.5, multiplying

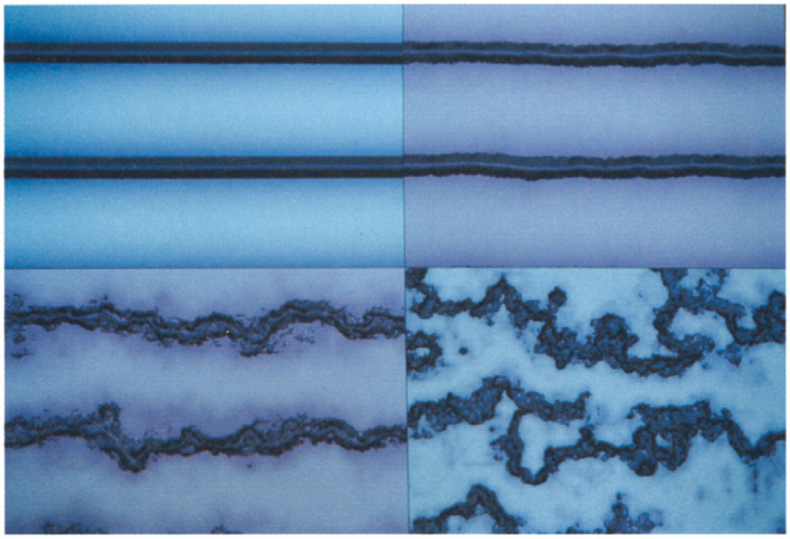

FIGURE 8.12 Liquid marble forming. Copyright © 1993 David S. Ebert.

by a scalar, adding 0.5, and limiting the result to the range of 0.2 to 1.0) is used to form a hard surface more quickly (Perlin 1992). The result of this function can be seen in Figure 8.12.

```
    //
    // parms[1] = maximum density value: density scaling factor
    // parms[2] = exponent for density scaling
    // parms[3] = x resolution for Perlin's trick (0-640)
    // parms[8] = 1/radius of fuzzy area for Perlin's trick(> 1.0)
    //
void molten_marble(xyz_td pnt, float *density, float *parms,
                   vol_td vol)
{
 float parms_scalar, turb_amount;
 turb_amount = solid_txt(pnt,vol);
 *density = (pow(turb_amount, parms[2]) )*0.35 +0.65;
 // Introduce harder surface more quickly.
 // parms[3] multiplied by 1/640
 *density *=parms[1];
 parms_scalar = (parms[3]*.0015625)*parms[8];
 *density= (*density-0.5)*parms_scalar +0.5;
 *density = MAX(0.2, MIN(1.0,*density));
}
```

PARTICLE SYSTEMS: ANOTHER PROCEDURAL ANIMATION TECHNIQUE

As previously mentioned, particle systems are different from the rest of the procedural techniques presented in this book in that their abstraction is in control of the animation and specification of the object. Particle systems were first used in computer graphics by Reeves (1983) to model a wall of fire for the movie *Star Trek II: The Wrath of Khan.* Since particle systems are a volumetric modeling technique, they are most commonly used to represent volumetric natural phenomena such as fire, water, clouds, snow, and rain (Reeves 1983). *Structured particle systems,* an extension of particle systems, have also been used to model grass and trees (Reeves and Blau 1985).

A particle system is defined by both a collection of geometric particles and the algorithms that govern their creation, movement, and death. Each geometric particle has several attributes, including its initial position, velocity, size, color, transparency, shape, and lifetime.

To create an animation of a particle system object, the following are performed at each time step (Reeves 1983):

- New particles are generated and assigned their attributes.
- Particles that have existed in the system past their lifetime are removed.
- Each remaining particle is moved and transformed by the particle system algorithms as prescribed by their individual attributes.
- These particles are rendered, using special-purpose rendering algorithms, to produce an image of the particle system.

The creation, death, and movement of particles are controlled by stochastic procedures, allowing complex, realistic motion to be created with a few parameters. The creation procedure for particles is controlled by parameters defining either the mean number of particles created at each time step and its variance or the mean number of particles created per unit of screen area at each time step and its variance. These values can be varied over time as well. The actual number of particles created is stochastically determined to be within *mean* + *variance* and *mean* − *variance.* The initial color, velocity, size, and transparency are also stochastically determined by mean and variance values. The initial shape of the particle system is defined by an origin, a region about this origin in which new generated particles are placed, angles defining the orientation of the particle system, and the initial direction of movement for the particles.

The movement of particles is also controlled by stochastic procedures (stochastically determined velocity vectors). These procedures move the particles by adding their velocity vector to their position vector. Random variations can be added to the velocity vector at each frame, and acceleration procedures can be incorporated to simulate effects such as gravity, vorticity, conservation of momentum and energy, wind fields, air resistance, attraction, repulsion, turbulence fields, and vortices. The simulation of physically based forces allows realistic motion and complex dynamics to be displayed by the particle system, while being controlled by only a few parameters. Besides the movement of particles, their color and transparency can also change dynamically to give more complex effects. The death of particles is controlled very simply by removing particles from the system whose lifetimes have expired or who have strayed more than a given distance from the origin of the particle system.

The Genesis Demo sequence from *Star Trek II: The Wrath of Khan* is an example of the effects achievable by such a particle system. For this effect, a two-level particle system was used to create the wall of fire. The first-level particle system generated concentric, expanding rings of particle systems on the planet's surface. The second-level particle system generated particles at each of these locations, simulating explosions. During the Genesis Demo sequence, the number of particles in the system ranged from several thousand initially to over 750,000 near the end.

Reeves extended the use of particle systems to model fields of grass and forests of trees, calling this new technique structured particle systems (Reeves and Blau 1985). In structured particle systems, the particles are no longer an independent collection of particles, but rather form a connected, cohesive three-dimensional object and have many complex relationships among themselves. Each particle represents an element of a tree (e.g., branch, leaf) or part of a blade of grass. These particle systems are, therefore, similar to L-systems and graftals, specifically probabilistic, context-sensitive L-systems. Each particle is similar to a letter in an L-system alphabet, and the procedures governing the generation, movement, and death of particles are similar to the production rules. However, they differ from L-systems in several ways. First, the goal of structured particle systems is to model the visual appearance of whole collections of trees and grass, and not to correctly model the detailed geometry of each plant. Second, they are not concerned with biological correctness or modeling the growth of plants. Structured particle systems construct trees by recursively generating subbranches, with stochastic variations of parameters such as branching angle, thickness, and placement within a value range for each type of tree.

Additional stochastic procedures are used for placement of the trees on the terrain, random warping of branches, and bending of branches to simulate tropism. A forest of such trees can, therefore, be specified with a few parameters for distribution of tree species and several parameters defining the mean values and variances for tree height, width, first branch height, length, angle, and thickness of each species.

Both regular particle systems and structured particle systems pose special rendering problems because of the large number of primitives. Regular particle systems have been rendered simply as point light sources (or linear light sources for anti-aliased moving particles) for fire effects, accumulating the contribution of each particle into the frame buffer and compositing the particle system image with the surface-rendered image. No occlusion or interparticle illumination is considered. Structured particle systems are much more difficult to render, and specialized probabilistic rendering algorithms have been developed to render them (Reeves and Blau 1985). Illumination, shadowing, and hidden surface calculations need to be performed for the particles. Since stochastically varying objects are being modeled, approximately correct rendering will provide sufficient realism. Probabilistic and approximate techniques are used to determine the shadowing and illumination of each tree element. The particle's distance into the tree from the light source determines its amount of diffuse shading and probability of having specular highlights. Self-shadowing is simulated by exponentially decreasing the ambient illumination as the particle's distance within the tree increases. External shadowing is also probabilistically calculated to simulate the shadowing of one tree by another tree. For hidden surface calculations, an initial depth sort of all trees and a painter's algorithm are used. Within each tree, again, a painter's algorithm is used, along with a back-to-front bucket sort of all the particles. This will not correctly solve the hidden surface problem in all cases, but will give realistic, approximately correct images. Figure 8.13 contains images from the animation *The Adventures of André & Wally B*, illustrating the power of structured particle systems and probabilistic rendering techniques for structured particle systems.

Efficient rendering of particle systems is still an open active research problem (e.g., Etzmuss, Eberhardt, and Hauth 2000). Although particle systems allow complex scenes to be specified with only a few parameters, they sometimes require rather slow, specialized rendering algorithms. Simulation of fluids (Miller and Pearce 1989), cloth (Breen, House, and Wozny 1994; Baraff and Witkin 1998; Plath 2000), and surface modeling with oriented particle systems (Szeliski and Tonnesen 1992) are recent, promising extensions of particle systems. Sims (1990) demonstrated the suitability of highly parallel computing architectures to particle systems simulation.

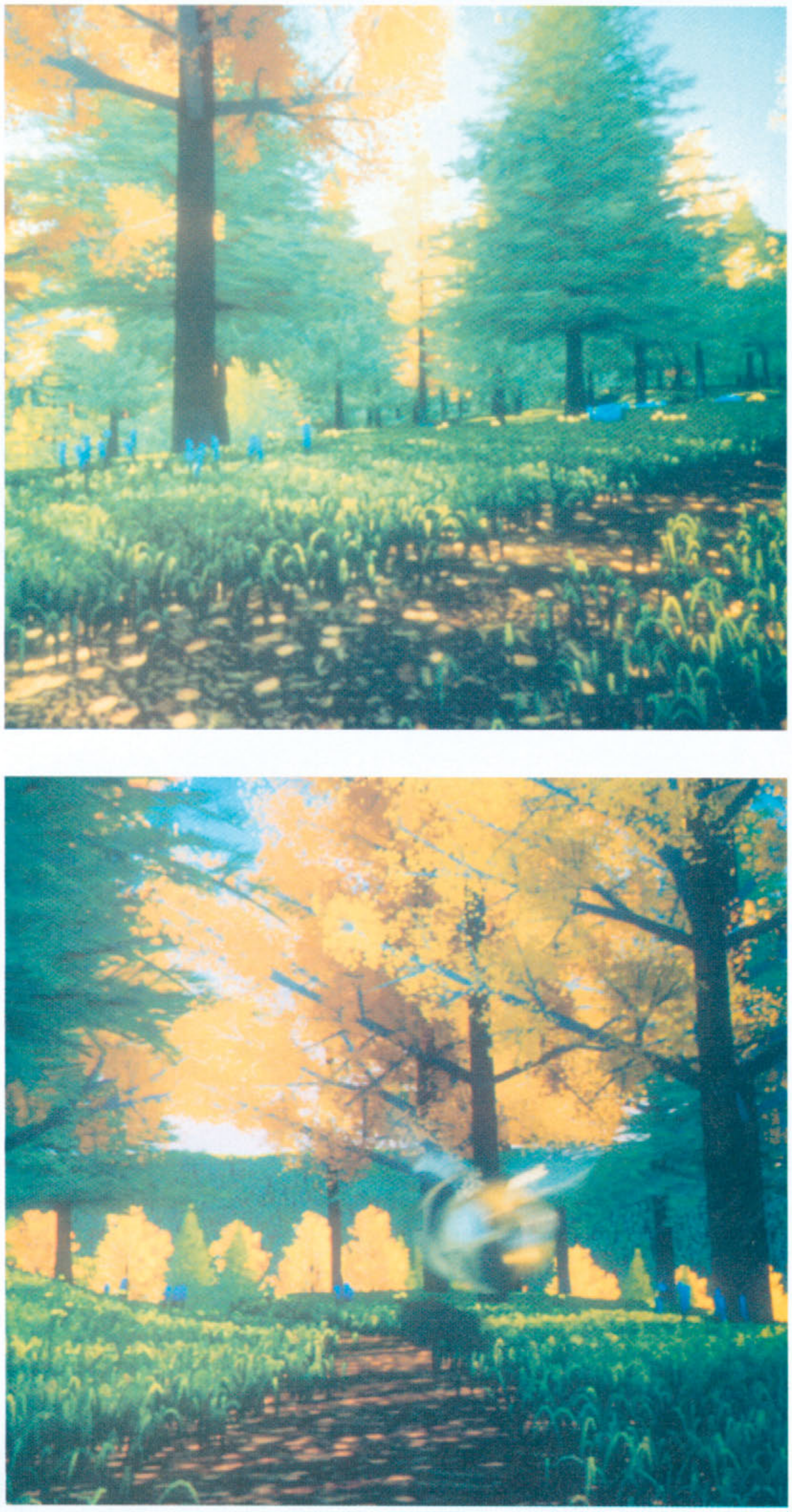

FIGURE 8.13 Examples of probabilistic rendering of structure particle systems from *The Adventures of André and Wally B.* Courtesy of W. Reeves and R. Blau.

Particle systems, with their ease of specification and good dynamical control, have great potential when combined with other modeling techniques such as implicit surfaces (Witkin and Heckbert 1994) and volumetric procedural modeling.

Particle systems provide a very nice, powerful animation system for high-level control of complex dynamics and can be combined with many of the procedural techniques described in this book. For instance, turbulence functions are often combined with particle systems.

CONCLUSION

This chapter described several approaches to animating procedural models and showed several practical examples. The general animation approaches presented can be used with any procedural model or texture. The next chapter discusses how hybrid procedural models can be used to model and animate procedural clouds, while hardware acceleration techniques for these animation approaches can be found in Chapter 10.