Rendering and Animation of Gaseous Phenomena by Combining Fast Volume and Scanline A-buffer Techniques

David S. Ebert and Richard E. Parent

Department of Computer and Information Science
The Ohio State University
2036 Neil Ave.
Columbus, Ohio 43210-1277

Abstract

This paper describes a new technique that efficiently combines volume rendering and scanline a-buffer techniques. This technique is useful for combining all types of volume-rendered objects with scanline rendered objects and is especially useful for rendering scenes containing gaseous phenomena such as clouds, fog, and smoke. The rendering and animation of these phenomena has been a difficult problem in computer graphics.

A new algorithm for realistically modeling and animating gaseous phenomena is presented, providing true three-dimensional volumes of gas. The gases are modeled using turbulent flow based solid texturing to define their geometry and are animated based on turbulent flow simulations. A low albedo illumination model is used that takes into consideration self-shadowing of the volumes.

CR Categories and Subject Description I.3.3 [Computer Graphics]: Picture/Image Generation - Display Algorithms; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism. **Additional Keywords:** volume rendering, a-buffer, gaseous phenomena, clouds, fog, solid texturing.

INTRODUCTION

The rendering of scenes containing clouds, fog, atmospheric dispersion effects, and other gaseous phenomena has received much attention in the computer graphics literature. Several papers deal mainly with atmospheric dispersion effects [20, 15, 18], while many cover the illumination of these gaseous phenomena in detail [1, 9, 14, 11]. Most authors have used a low albedo reflection model, while a few, Blinn [1], Kajiya [9], and Rushmeier [18], discuss the implementation of a high albedo model.

A major shortcoming of previous efforts in rendering scenes containing gaseous phenomena is that they have required that the same rendering techniques be used for all elements in the scene. This has made animations containing gaseous phenomena very computationally expensive and severely limited their usefulness. Kajiya requires that everything be raytraced, which makes the rendering of other objects in the scene very slow [9]. The approach of Rushmeier requires a radiosity solution of the scene which yields a very accurate illumination model, but at a high computational cost [18].

Most volume rendering systems also require that everything in the scene be volume rendered. Kaufman [10] allows the combination of surface geometry-defined objects with volume defined objects, but requires that the surface geometry-defined objects be converted to volumes before rendering, which is very inefficient. Levoy has used a similar approach to combine polygonal and volume data [12]. One volume rendering system that does not require that everything be volume rendered is described in [13]. This system uses raytracing for polygonal data.

One solution to these problems is image compositing of scenes [3]. This solution would take volume-rendered objects and normally-rendered objects and combine them in a post-process. However, this is a limited solution, with only a few cases resulting in an accurate image. When compositing the sub-images, the volume-rendered image cannot cast shadows on the objects in the other scene. Also, compositing of an image containing semi-transparent volume-rendered objects with another image can only provide satisfactory results when the volume rendered object in the first image is totally in front of or behind the objects in the other image. For example, compositing of volume-rendered fog with a street scene is not possible. This suggests that to get accurate images of scenes containing gaseous phenomena, image compositing techniques are not a suitable solution.

The technique described in this paper solves these problems by allowing a fast scanline a-buffer technique to be used to render objects described by surface geometries, while volume modeled objects are volume rendered. This technique even allows the volume modeled objects to accurately cast shadows on the other objects in the scene.

Another issue is modeling the geometry of these gases. Some authors use a constant density medium [11, 15], but do allow different layers of constant densities. To model the geometry of clouds, Kajiya uses a physically-based model [9], Gardner uses hollow ellipsoids [7], Voss uses fractals [19], and Max uses height fields [14]. This paper shows that realistic results can be obtained by using turbulent flow based functions to model the density of a variety of gases. These functions are based on Perlin's visual simulation of turbulent flow [16] and are similar to the idea of hypertextures [17].

The techniques described in this paper seem to provide more realistic results than most previous efforts by providing visually realistic renderings and animations of gaseous phenomena and the shadows they cast. These techniques are based on a visual simulation of turbulent flow, so it is a visual simulation of the turbulent

processes that determines the geometry of gaseous phenomena. These techniques can also be extended to use a physically-based turbulent flow model and can be very efficient when simplifying assumptions are made.

In this paper, we discuss the efficient combination of volume rendering and a-buffer rendering techniques to provide realistic animations of gaseous phenomena. The algorithm which combines these techniques is discussed first. From a review of the literature, it appears that this is the first system that allows the combination of volume-rendered and scanline-rendered objects in the same scene. The following sections describe the volume rendering, illumination and modeling of the gases. Finally, a discussion of the realistic results obtainable by these techniques is presented along with future extensions.

Combining A-buffer and Volume Rendering Techniques

The rendering system described in this paper efficiently combines scanline a-buffer rendering with volume rendering without any restrictions on the geometric positioning or overlap of the volume and non-volume elements. 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 that pixel. At this point, the illumination calculations for the fragments have not been done.

The fragment structure is similar to that used by Carpenter in his original paper on the a-buffer [2]. The structure used is as follows:

Fragment Structure
- minimum and maximum z values - percent coverage - normal vector - pointer to parent object - bitmask for geometry of coverage - color - light attenuation values for each light

The additional information added when combining volume rendering with the a-buffer technique is an attenuation amount for each light source, which is used for shadowing of the fragment by the volume elements. In the system described in this paper, the volumes are defined by procedural functions. The data structure used for visibility calculations of the volume modeled objects is the a-buffer fragment structure described above. The following section describes how the volume modeled objects are broken into sections to create a-buffer fragments and combined with the surface-defined a-buffer fragments.

If a volume element is active for a pixel, the extent of volume tracing that is required is determined in the following manner. First, the ray from the eye through the pixel projected into world space is calculated. Then each of the scanline-rendered a-buffer fragments are mapped back to world space. From the geometry of the volume, the position of these a-buffer fragments in world space, and the hither and yon planes for the scene, the extent of the volume that must be traced can be computed. The starting point for the volume tracing is the maximum of the hither plane and the closest point of intersection of the ray with the volume. The end point for the volume tracing is determined by the intersections of the ray with the volume, the yon plane, and the a-buffer fragments for this pixel. The a-buffer fragment list is traversed to determine the location in world space where full coverage of the pixel is obtained. Volume rendering will terminate at the minimum of this location, the yon plane, and the farthest intersection point of the ray and the volume.

Each fragment in the fragment list also determines a starting or stopping point for separate volume elements to be added into the a-buffer fragment list. To get correct effects, the volume to be rendered must be broken into sections that lie between, in front of, and behind the a-buffer fragments (see Figure 1). For example, if there is a transparent object covering a pixel containing fog, a separate a-buffer fragment must be created for the fog in front of the transparent object, fog inside the transparent object (if desired), and the fog behind the transparent object. This is necessary since, if only one fragment was created, the fragment would be sorted in front of, in, or behind the transparent object's fragments based on the average z value of the fog, resulting in an incorrect pixel value.

The volume rendering for this pixel then takes place, creating new fragments for the a-buffer fragment list that are sorted into place. For each of the fragments, computations to determine the shadowing by the volumes are performed.

After the volume rendering is complete for each pixel, the geometries of the bitmasks are used to determine the visible fragments for the pixel. This is done similar to [2] except that the illumination, texturing, etc., calculations are only performed for visible fragments. For volume fragments, the bitmask will contain all 1s since only one ray is traced per pixel for volume elements. From the authors' experience, this resolution, although lower than that for surface-defined objects, provides quality images.

The only additional computations performed for scanline-rendered fragments in combining these techniques is the inverse mapping of the fragment to world space and the shadow tracing through the volume. This inverse mapping needs to be done for most types of texturing, so this is not an additional expense if the world space coordinates of the points are saved in the fragment structure. These shadow tracing calculations are needed for accurate shadowing. There is very little additional expense for scanline-rendered fragments when these techniques are combined in this way. Figure 3 shows an image of an art gallery, containing a volume modeled gaseous object as part of a modern sculpture. This scene was rendered with and without the volume modeled object. Calculation times are given in the results section. It will become clear from these timings that the overhead of combining the rendering techniques is negligible. The majority of rendering time is spent in the texturing of all the different objects.

Volume rendering

As mentioned above, only the visible portion of the volume is rendered. Volume tracing stops once full coverage of the pixel is reached. If the volume is completely covered by a wall in the scene, for example, no volume rendering takes place. Depending on scene composition, this can have significant time savings. The volume rendering technique used here is similar to the one discussed in [17]. The ray from the eye through the pixel is traced through the defining geometry of the volume. As described above, volume tracing stops once full coverage of the pixel is obtained. If there is partial coverage of the pixel by some fragments, the volume is broken into sections in front of and behind the fragments until full coverage is reached. It is necessary to do this because each of the sections becomes a new element on the a-buffer fragment list. For each increment through the volume, the density function is evaluated. If the volume density function represents a solid object, like the hypertexture functions described in [17], a slightly different algorithm is used than if the volume density functions represent a gas. The two algorithms differ in their illumination calculations and accumulation of densities. If normal illumination techniques are going to be used (for solid volumes), the normal to the surface is also calculated by the method described by Perlin [17]. If a gaseous illumination model is to be used these additional functional evaluations are not needed. The densities are also accumulated differently depending on whether the volume is a gas or a solid.

The basic algorithm for rendering solid volumes is the following:

```
determine 2 mutually orthogonal directions
  to the ray
for each section of volume
   get the density value at the previous point
   for each increment along the ray
      get color, density, & opacity of this element
      get  density in 2 mutually orthogonal
         directions
      determine the normal to the surface based
         on previous density, current density
         density in dir1,density in dir2
      if self_shadowing
        for each light source
        trace the ray to the light getting
             the light attenuation factor
       color = calculate the illumination of
             this volume using this normal and
             the color of the element
       t1 = opacity*(1-sum_opacity);
       final_clr = final_clr + t1*color;
       sum_opacity =sum_opacity +t1;
       if sum_opacity =1
          stop tracing
       increment the sample_pt
       previous density = density
   create the a_buffer fragment
```

The opacity is determined from the density functions using the following formula from [17]:

$$opacity = 1 - (1 - density)^{c*step\_size}$$

where c is a normalizing constant used to make opacity a function of both density and step size.

The algorithm for rendering gaseous volumes is the following:

```
for each section of gas
   for each increment along the ray
      get color, density, & opacity of this element
      if self_shadowing
        for each light source
           trace the ray to light getting the light
                attenuation factor
        color =calculate the illumination of 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
```

Here, the opacity is the value returned from evaluating the density function multiplied by the step-size. This is needed since in the gaseous model, we are approximating an integral to calculate the opacity along the ray. The integral from [9] is

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

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.

This is being approximated by

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

As Kajiya suggests [8], the final increment along the ray may not be the same size as the rest, so its opacity is scaled proportionally.

The system currently implemented renders function-based volume densities, but can easily be extended to handle voxel-based volume objects. In sampling along the ray, a Monte-Carlo method is used in choosing the point.

Illumination of gaseous Phenomena

The illumination algorithm that is used is based on [9]. We have implemented the low-albedo illumination model. The phase-functions that are used are sums of Henyey-Greenstein functions as described in [1]. The illumination model is the following:

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

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 [1]. $I_i(x(t),y(t),z(t))$ is the amount of light from light source i reflected from this element. Self-shadowing of the gas is incorporated in this term by attenuating the brightness of the light. To approximate a high albedo model, an ambient term based on the albedo of the material can be added into I_i. This ambient term accounts for the percentage of light reflected from the element due to second and higher order scattering effects.

Shadowing of the gas

The simplest way of shadowing the gas is to trace a ray from each of the volume elements to be rendered to the light, determining the opacity of the material along the way using the above equation for opacity. This method is similar to shadowing calculations performed in ray tracing and can be very slow. Depending on scene composition (amount of gas in the scene), our experiments have shown that self-shadowing in this manner can account for 75-95% of the total computation time.

Kajiya talks of the importance of self-shadowing to correctly visualize data [8]. However, he shows that low-albedo models for gases with albedos over 30% produce too much self-shadowing [9]. If the gas to be rendered has a very high albedo, the effects of self-shadowing are negligible compared to the secondary and higher order scattering of light. Thus for gases with a high albedo, not performing self-shadowing can give realistic results at a much lower computational expense. For patchy fog or other gases with gaps of low density, shadowing can be sped up by not calculating shadows for elements where the density is less than a threshold value.

In order 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 [9, 8] . This restriction was necessary for the method in which the table was produced, but is removed in the technique we propose in this paper. Using this technique, the light source may even be inside the volume. Use of a precalculated table is definitely faster for gases that do not move from one frame to the next. However, even if the gas does move from one frame to the next, it still provides faster rendering. The main reason that the shadow tracing calculations account for so much calculation time is that a large percentage of the calculations are repeated. In determining the shadowing for point p_i, the ray from p_i to each light source is traced through the volume. While tracing this ray, the shadowing information for some points are also calculated, but not stored; therefore, shadowing calculations are repeated for these points. Shadowing calculations for points in the volume near the light source may be performed many times, depending on the order of processing. The amount of repeated calculations can be seen from a simple example. Assume a cubic volume of gas surrounding the entire scene, an image size of

500x500, a sample size of 1/40th the volume size, the observer located along the positive Z axis, and the light source located directly above the volume. In determining the geometry of the gas, 500x500x40 = 10,000,000 volume-density function calculations will be performed. To calculate the shadow value for each of the elements, an average of 20 volume-density function calculations will need to be performed, for a total of 200,000,000 volume-density function calculations for shadowing.

The obvious way to eliminate these repeated calculations is to store all the shadow values that are calculated in a large three-dimensional table. Then, when a ray is traced towards the light, if the shadow value has already been calculated for the point currently being sampled, the shadow tracing stops and this shadow value is added to the shadow values already accumulated. The main reason this approach is infeasible is the size of the table. If the image size is 640x480 and 40 samples deep are being made, 12 Mb of memory would be required to store the table if only shadow values between 0 and 255 are being stored. However, this approach would become feasible if a reduced-resolution table were used, which is the approach the authors have chosen.

Calculation of the shadow table

To calculate the reduced-resolution shadow table values, a table of the same dimensions containing functional values is first computed. This is done to avoid repeated density functional evaluations. (If these functional evaluations are faster than a bilinear interpolation, this step would not save any time.) Next, the distance squared from each point in the table to the light is calculated. These distances are then sorted, providing the order in which the shadow table values should be calculated. By calculating the shadow table values starting with the points closest to the light and proceeding to the points farthest from the light, only a bilinear interpolation is needed to determine each value. In determining the shadow value for a table entry, the ray to the light from this element is calculated (see figure 2). The ray from the point, $p_{i,j,k}$ to the light will pass though one of the faces of a parallelepiped formed by table entries (i+1,j+1,k+1), (i+1,j-1,k+1), (i+1,j-1,k-1), (i-1,j+1,k+1),(i-1,j-1,k+1), (i-1,j-1,k-1). Now using the step sizes between table elements in world space and the normalized vector to the light, the face of the cube that surrounds this table element which will be pierced by the ray to the light can be determined. Once this is determined, the point of intersection of the ray and that plane can very quickly be determined since the table is aligned with the axes in world space. The restriction that the table aligns with the axes could easily be removed at the expense of some additional computations. This point of intersection now lies between 4 table entries that already contain the shadow information for those points since the entries are calculated in sorted order. By adding the functional values of the entries times the step size to their shadow table values and bilinearly interpolating these sums, the shadow value for the current element can be found.

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 8 entries are tri-linearly interpolated to obtain the sum of the densities between this sample point and the light. To determine the amount to attenuate the light, the following formula is used.

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

This method is faster than tracing rays to each light if multiple volume density functional evaluations are slower than a tri-linear interpolation plus a fraction of the time needed to create the table. The average number of functional evaluations that are needed in determining the shadowing of an element by tracing the ray to the light will depend on the step size chosen for the volume tracing (normally tens of functional evaluations). The functions that the authors use are based on turbulent flow and are much slower than the above. These functions will be discussed in a later section. Time comparisons can be found in the results section.

MODELING AND ANIMATING THE GASES

Surface defined gases

The authors originally used solid texturing of polygonal mesh objects for modeling gaseous phenomena [6]. The solid texturing functions are based on the turbulence function from [16]. To simulate gaseous phenomena, solid texturing was used to control the transparency of objects that define the space that the gaseous substance occupies. In this way, solid textured transparency can be used to simulate fog, smoke, and clouds. For a cloud layer, one flat plane can be used. The transparency of this plane creates the clouds. The transparency is controlled by a turbulent flow based solid texturing function. A similar result could be obtained by having solid textured ellipsoids occupying the same solid texturing space. By using the ellipsoids, it is possible to get results similar to Gardner [7]. Gardner uses flat planes for creating cloud layers and ellipsoids for creating individual clouds positioned in space.

A simple function to generate clouds is the following:

```
clouds(pnt,pixel_size,frequency,power)
    xyz_td  pnt; /* the location of the point in
                  * the solid texture space */
    float   pixel_size, frequency,power;
{
  /* add some noise to the pnt */
 pnt.x += noise(pnt); pnt.y += noise(pnt);
  /* use a sine wave for the basis of the shape */
 tmp=sin((turbulence(pnt, pixel_size)*frequency));
 return(1.0 -pow(tmp+1.0,power)*.5);
}
```

By changing the parameters in this function, static images of fog, mist, or smoke can be created. The *frequency* parameter controls the frequency of the sine wave through the turbulent space. The *power* parameter controls the distribution of the transparency values, basically determining whether linear interpolation, quadratic interpolation, or other types of interpolation are used to generate the transparency value. The other parameter that greatly affects the shape of the gaseous substance is the size of the solid texture space relative to the shape-defining object. Increasing this relative size creates the effect of zooming in on a portion of the gas.

Volume-modeled gaseous phenomena

The above technique has been extended to volume modeling of the gaseous phenomena. The turbulent flow based functions now control the density and geometry of the gases rather than the transparency of a polygonal object. In volume modeling these phenomena, a 3D solid defines the space that these gases occupy. Any ray-traceable solid can be used. One problem with this approach is that the solid's boundaries can be observed. However, by controlling the density functions, the defining shape of the solid can be made undetectable (if desired). This is easily accomplished by decreasing the density based on the location within the volume.

Animating the gaseous phenomena

There are two obvious ways that the above technique which simulates gaseous phenomena can be animated. The first technique animates the turbulent space. The second technique moves the objects through the turbulent space. The first would be better for a true physically-based simulation of turbulent flow. However,

since the method described here is not a physically-based simulation of turbulent flow, animating the turbulent space would be artificial and not worth the computational expense. The second, moving the objects through the turbulent space, is the approach described here since it is computationally more efficient and still gives effective results.

A surface based version of animating solid textured gases was implemented to create the swirling fog in the animation "Once a Pawn A Foggy Knight..." [4]. A sample image from this film is in figure 4. Three planes are positioned in the scene and are used to produce different movement patterns in the fog giving a more effective depth to the fog. Each plane has different parameters to the fog function to create varying amounts of transparency and motion. More planes would produce more complex motion but greatly increase the computational expense. These planes all reside in a large solid texture space that surrounded the entire scene. The turbulence function is defined over this solid textured space. Each point on these planes that corresponds to a pixel location to be rendered is moved through this turbulent space based on the frame number and other parameters of the function for each plane. To animate the fog, each of these points is moved along a helical path through the solid texture space. Each point on the plane is then actually tracing out a different 3D path over time. Since the points on the plane are moving through the solid space, the fog appears to be 3D since you are seeing different points of the three dimensional volume of fog over time.

A similar technique is used to animate full three dimensional volume modeled gaseous phenomena. Instead of using multiple planes, as mentioned earlier, a solid object is used to define the space the gas occupies. In this case, each point in the volume is moved along a helical path to provide swirling gases. Figure 6 shows frames from a animation featuring volume-modeled steam rising from a glass. Notice the shadows cast on the wall and on the inside of the glass by the steam. The steam itself has no self-shadowing.

Many effects can easily be created from these simple functions. In the figure, the steam dissipates as it moves farther from the glass. This dissipating effect is created by decreasing the opacity as a function of the distance from the glass. The results section contains information about resolution and calculation times for the figures. Figure 7 shows frames from an animation featuring volume-modeled fog rolling in from the left of the screen. This also shows a chess piece moving through the fog. This attests to the true three dimensionality of the fog. To create the effect of fog moving in, each point in the volume was compared to a value, f, indicating the front of the fog. If the point's x value was greater than f, the density was set to 0. f changes based on the frame number, the turbulence value of the point, and the three dimensional location of the point in world space. This is done so that the front of the fog does not appear as a plane. It is actually a deformed plane where the deformation is controlled by turbulence and a table of random numbers. To create the fog thickening, the maximum density of the fog increases from zero at the front of the fog to one after a certain distance. The maximum density of the fog could also be increased based on the frame number so that the fog grows from thin wisps to dense patches. To create these effects, functions similar to the following are used. This function will create fog whose density increases over time and moves horizontally.

```
chess_fog(pnt,density,pixel_size,parms, pnt_w,vol)
     xyz_td  pnt, pnt_w;
     float   *density,*parms, pixel_size;
     vol_td  vol;
{
 float tmp, factor, front_of_fog, factor2;
 extern int frame_num;
 extern float offset[OFFSET_SIZE];
 xyz_td direction,cyl;
 int    indx;

 /* apply some turbulence to the point
  * so that it appears more random */
 tmp = turbulence(pnt, pixel_size);
 pnt.x += 2.0 + tmp;
 pnt.y += .5 +tmp;
 pnt.z += -2.0 - tmp;

 /* determine how to move the point through
  * the space (helical path)  */
 theta =(frame_num%SWIRL_FRAMES)*SWIRL;          (1)
 cyl.y = ELLIPSE1 * (float)cos(theta);
 cyl.z = ELLIPSE2 * (float)sin(theta);
 direction.x = pnt.x - (float)frame_num*DOWN;
 direction.y = pnt.x + cyl.y;
 direction.z = pnt.z + cyl.z;
  /* now determine the points transparency based
   * on its new location in the solid space */
 tmp =turbulence(direction, pixel_size);
  /* have the fog grow more dense */
 factor = MIN((frame_num/FULL_FOG.0),1.0);
 *density =  (tmp*opaque*factor);                (2)
 *density = pow(*density, power)                 (3)
}
```

The section of code (1) calculates how to move the point along the helical path based on the frame number, the number of frames to trace out an ellipse in the y-z plane (SWIRL_FRAMES and SWIRL), the amount to move horizontally per frame (DOWN), and the major and minor diameters of the ellipse (ELLIPSE1, ELLIPSE2). These parameters control the amount and direction of the overall fog movement. The intricacies in the fog movement are created by the turbulence function. The parameters, *opaque* and *power*, in transparency calculation (2) and (3) are used to control the opaqueness of the fog. Adjusting these values can change the density distribution of the fog since it shifts the opaqueness levels. The *factor* causes the density to increase from a maximum value of 0 to a maximum value of opaque over *FULL_FOG* frames. Varying *opaque* can be used to to get patchy fog as opposed to wispy fog.

Use of turbulent flow for gases

Using turbulent flow based functions for the simulation of fog makes sense since the varying densities of the fog are created by turbulent flow. Turbulence also creates the path over which the fog moves. Currently, a visual simulation of turbulent flow is used. This system can easily be extended to use a physical-based simulation of turbulent flow, since all that is needed is a different function to determine the density based on atmospheric turbulent flow. Developing the function to model atmospheric turbulent flow is a separate nontrivial problem.

DISCUSSION OF RESULTS

This paper has shown that volume rendering and scanline a-buffer rendering can be efficiently combined. This new technique is very useful when combined with turbulent flow based volume-modeled gases to produce realistic scenes with gaseous phenomena. All results were calculated on a Hewlett-Packard 9000/370 TSRX workstation with 16 Mb of memory. Table 1 shows rendering times for Figures 3 through 7. Table 2 shows rendering times for each of the images in Figure 8. All rendering times are approximate. Figure 3 shows an art gallery scene containing a volume rendered gaseous sphere without volume shadowing. Table 1 provides calculation times for the image in figure 3 and the same scene without the volume. From these times, it is clear that the addition of a volume rendered object in the scene only slightly increases the computation time.

Figure 4 shows a scene from "Once a Pawn a Foggy Knight..." showing surface based solid texturing to create the fog. Figure 5

shows the results obtainable by extending the turbulent flow based solid texturing to volume modeling. This scene is from "Getting Into Art" [5], and has table-based volume shadowing. Figure 6 shows 6 images of volume-modeled steam rising out of a glass casting shadow-traced shadows on the glass and the wall. The steam, however, does not shadow itself. These images represent every 30th frame of an animated sequence. Figure 7 shows 9 frames of a chess scene similar to figure 4. In these images, the fog is volume rendered and is "rolling" in from the left of the image. Notice the shadows of the fog on the rook change as it moves forward. Also notice the shadows of the fog move across the curved base of the rook, the top of the pawn, and the ground plane in the last 3 images.

Figure	*Resolution*	*Rendering Time (minutes)*	
3	1024 x 682	71	with volume (actual picture)
	1024 x 682	59	without volume
4	1024 x 768	105	
5	1024 x 682	260	
6	420 x 400	95	each image
7	320 x 240	165	each image

Table 1: Rendering Times for Figures 3 through 7

From this discussion of the results, it is clear that the computational time increases proportionally to the amount of gaseous volume in the image. Self-shadowing of the gases is also very expensive if shadow tracing is used. Figure 8 shows 4 images of a glass with steam rising from it. The first image has no shadowing at all. The second image has shadowing of the volume onto the glass. The third and fourth images have self-shadowing of the volume and shadowing of the volume on the glass. In the third image, the shadow-table technique was used with shadow table dimensions of 64x64x64. In the fourth image, shadow-tracing was used. The calculation times are given in Table 2. This further illustrates the improvement in computational time with table-based shadowing. In the times for images with shadow-table based shadowing, the times include the time to create the shadow table.

Figure	*Shadowing Level*	*Rendering Time (minutes)*
a	none	80
b	shadow-traced, no self-shadowing	101
c	shadow-traced, with self-shadowing	1800
d	table-based, with self-shadowing	127

Table 2: Rendering Times for Figure 8. Resolution is 360x250

The above calculation times are an upper limit on the computation time required to get the results seen in the figures. The same visual results might be achieved by using larger step sizes in volume tracing and shadow tracing and by using lower resolution shadow-tables, which would decrease the computation time.

The combination of the volume rendering and scanline a-buffer rendering has wide ranging applications beyond the rendering of gaseous phenomena. This technique allows any volume data to be rendered in scenes with traditional surface based objects. The use of an a-buffer scanline renderer is not necessary if anti-aliasing is not desired. A normal scanline z-buffer could be used instead of the a-buffer; however, transparency will also not be handled easily, which is a normal problem with scanline z-buffer renderers.

Many additional performance improvements are possible. The authors found that initially 65% of computation time was being spent in turbulence function evaluations. We have since achieved a 60% performance improvement in the turbulence function evaluation alone. There are certainly more performance improvements possible in the volume rendering section of this new system. Another way to improve performance is to use a stored table of functional values with a lower resolution. This should provide improvements in computation time since volume density functional evaluations are the major computational expense.

Animation using the above techniques is very suitable for distributed processing. The authors have used over 150 SUN and Hewlett-Packard workstations in computing the animation "Once a Pawn a Foggy Knight... ." The same network distribution software used for this animation has been changed to handle the new techniques described above. Since the rendering time of a single frame is high, each workstation could also be given a range of scanlines to compute. The network distribution software took approximately 20 minutes to distribute jobs to 150 machines. This time could easily be decreased if detailed information about the performance of machines was not kept. By using a large distributed network like this, high resolution animations could easily be produced at the rate of more than 10 seconds per day.

Future Extensions

One extension to the above work is to extend the volume renderer to voxel-based volume rendering. This proposes no new problems in combining the volume rendering with the a-buffer.

The authors are interested in extending the techniques for rendering and animating the gaseous phenomena to be physically-based. One idea is to control the overall fog movement based on a more global turbulent directional field. In this way, for example, the fog would move towards areas of low turbulence. A table containing barometric pressure readings could also be used to create the directional movement of the gas by using the gradients of the values to control the speed and direction of the movement. In this way, you could begin to develop a physically-based model. The next step towards a physically-based model is to develop a physically-based turbulent flow model for various types of gases. This model seems to be computationally complex, but could easily be added into the current system since it is functionally based. As Perlin discusses, it is interesting how realistic the results obtained from a visual simulation of turbulent flow look [17]. The authors feel that developing a physically-based turbulent flow function, then simplifying the calculations to be tractable while maintaining the same visual quality is a good direction to take.

Acknowledgments

The authors wish to thank Keith Boyer, Jeff Ely, Bob Manson, and Rob Rosenblum for help in optimizing the turbulence functions. We also wish to thank Keith Boyer for writing the network distribution software, Wayne Carlson and Julia Ebert for reviewing this paper, Ed Tripp for design help, Jim Kent for slab intersection software, and the computer staff of the Department of Computer and Information Science for providing a stable computer environment with high availability. The authors also wish to thank Hewlett-Packard for their donation of equipment to the graphics lab.

References

[1] BLINN, JAMES. Light Reflection Functions for Simulation of Clouds and Dusty Surfaces. Proceedings of SIGGRAPH'82 (Boston, Massachusetts, July 26-30,1982). In *Computer Graphics 16,3* (July 1982), 21–29.

[2] CARPENTER, LOREN. The A-buffer, an Antialiased Hidden Surface Method. Proceedings of SIGGRAPH'84 (Minneapo-

lis, Minnesota, July 23-27, 1984). In *Computer Graphics 18, 3* (July 1984), 103–108.

[3] DUFF, THOMAS. Compositing 3-D Rendered Images. Proceedings of SIGGRAPH'85 (San Francisco, California, July 22-26,1985). In *Computer Graphics 19, 3* (July 1985), 41–44.

[4] EBERT, DAVID, BOYER, KEITH, AND ROBLE, DOUG. Once a Pawn a Foggy Knight ... [videotape]. In *SIGGRAPH Video Review 54* (November 1989), ACM SIGGRAPH, New York. segment 3.

[5] EBERT, DAVID, EBERT, JULIA, AND BOYER, KEITH. Getting Into Art. [videotape], Department of Computer and Information Science, The Ohio State University, May 1990.

[6] EBERT, DAVID, AND PARENT, RICHARD. Animation of gaseous phenomena using turbulent flow based solid texturing. Tech. Rep. OSU-CISRC-7/89-TR36, Department of Computer and Information Science, The Ohio State University, 2036 Neil Ave, Columbus, Ohio 43210-1277, July 1989.

[7] GARDNER, GEOFFREY. Visual Simulation of Clouds. Proceedings of SIGGRAPH'85 (San Francisco, California, July 22-26,1985). In *Computer Graphics 19, 3* (July 1985), 297–303.

[8] KAJIYA, JAMES, AND KAY, TIMOTHY. Rendering Fur with Three Dimensional Textures. Proceedings of SIGGRAPH'89 (Boston, Massachusetts, July 31-Aug 4,1989). In *Computer Graphics 23,3* (July 1989), 271–280.

[9] KAJIYA, JAMES, AND VON HERZEN, BRIAN. Ray Tracing Volume Densities. Proceedings of SIGGRAPH'84 (Minneapolis, Minnesota, July 23-27,1984). In *Computer Graphics 18,3* (July 1984), 165–174.

[10] KAUFMAN, ARIE. Efficient Algorithms for 3D Scan-Conversion of Parametric Curves, Surfaces, and Volumes. Proceedings of SIGGRAPH'87 (Anaheim, California, July 27-31,1987). In *Computer Graphics 21,4* (July 1987), 171–180.

[11] KLASSEN, R. VICTOR. Modeling the Effect of the Atmosphere on Light. *ACM Transaction on Graphics 6*, 3 (July 1987), 215–237.

[12] LEVOY, MARC. Private Communication, April 1990.

[13] LEVOY, MARC. A Hybrid Ray Tracer for Rendering Polygon and Volume Data. *IEEE Computer Graphics and Applications 10*, 2 (March 1990), 33–40.

[14] MAX, NELSON. Light Diffusion Through Clouds and Haze. *Computer Vision, Graphics, and Image Processing 33* (1986), 280–292.

[15] NISHITA, TOMOYUKI, MIYAWAKI, YASUHIRO, AND NAKAMAE, EIHACHIRO. A Shading Model for Atmospheric Scattering Considering Luminous Intensity Distribution of Light Sources. Proceedings of SIGGRAPH'87 (Anaheim, California, July 27-31,1987). In *Computer Graphics 21,4* (July 1987), 303–310.

[16] PERLIN, KEN. An Image Synthesizer. Proceedings of SIGGRAPH'85 (San Francisco, California, July 22-26,1985). In *Computer Graphics 19,3* (July 1985), 287–296.

[17] PERLIN, KEN, AND HOFFERT, ERIC. Hypertexture. Proceedings of SIGGRAPH'89,(Boston, Massachusetts, July 31-Aug 4,1989). In *Computer Graphics 20,3* (July 1989), 253–262.

[18] RUSHMEIER, HOLLY, AND TORRANCE, KEN. The Zonal Method for Calculating Light Intensities in the Presence of a Participating Medium. Proceedings of SIGGRAPH'87 (Anaheim, California, July 27-31,1987). In *Computer Graphics 21,4* (July 1987), 293–302.

[19] VOSS, RICHARD. Fourier Synthesis of Gaussian Fractals: 1/f noises, landscapes, and flakes. In *SIGGRAPH 83:Tutorial on State of the Art Image Synthesis* (1983), vol. 10, ACM SIGGRAPH.

[20] WILLIS, P.J. Visual Simulation of Atmospheric Haze. *Computer Graphics Forum 6* (1987), 35–42.

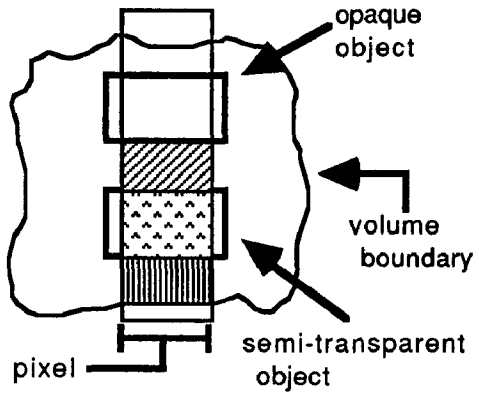

Figure 1: Volume element creation. Shaded areas are the three volume elements.

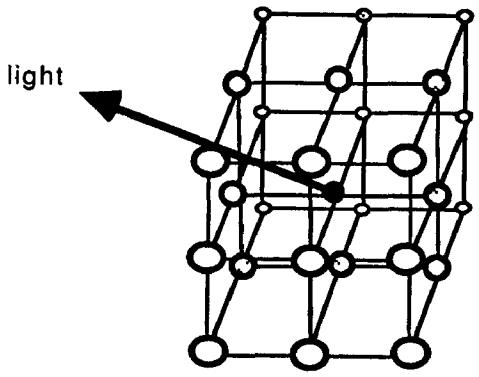

Figure 2: Shadow table Calculation.

Figure 3: The modern pioneer in an art gallery.

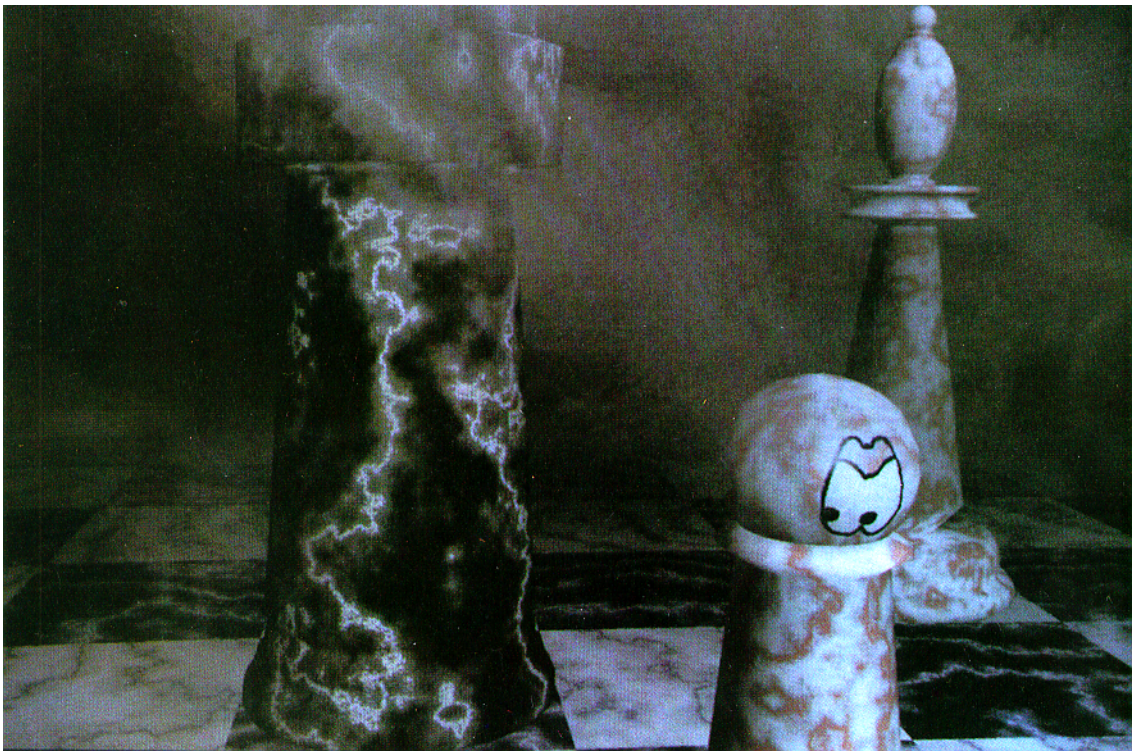

Figure 4: A representative image from "Once a Pawn a Foggy Knight...".

Figure 5: A representative image from "Getting Into Art."

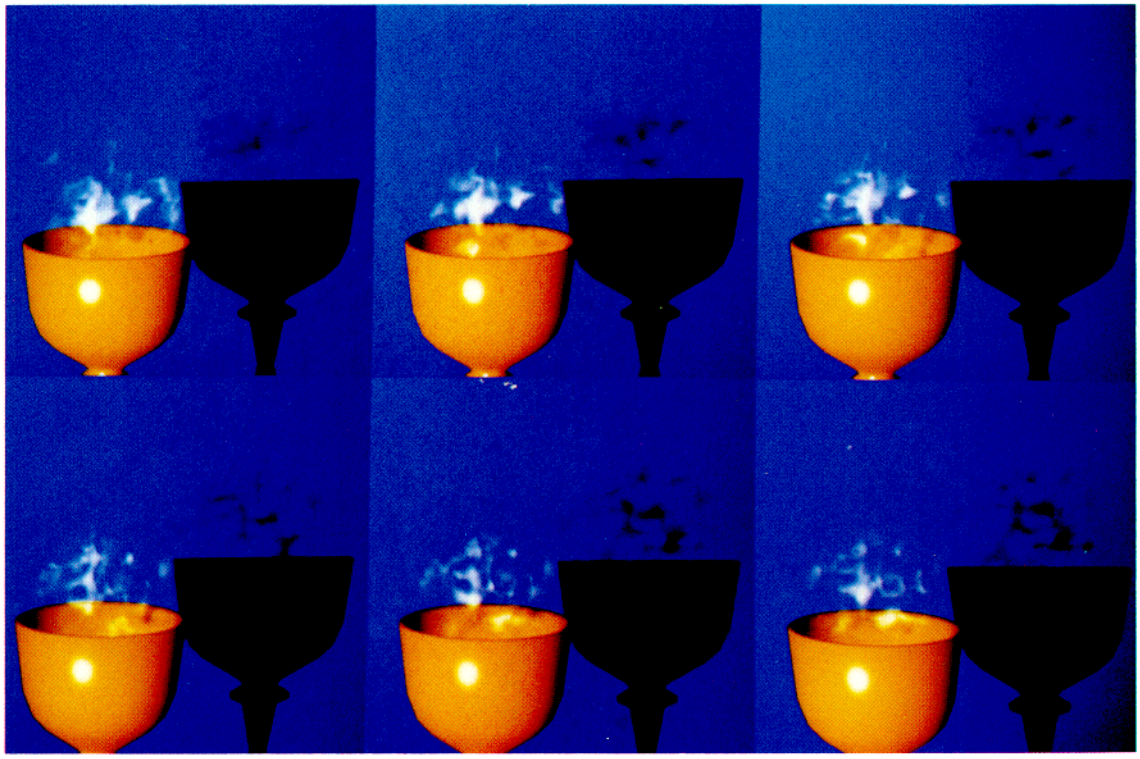

Figure 6: Steam rising from a glass. The 6 images are every 30 frames.

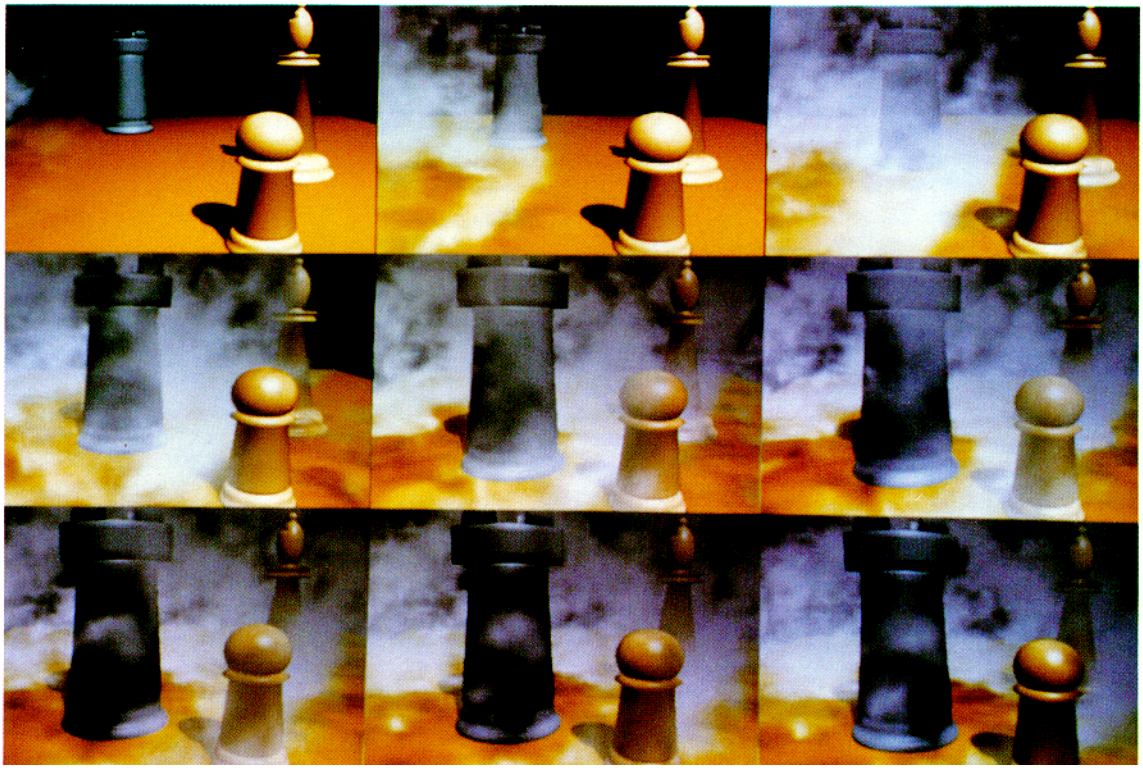

Figure 7: Scenes of fog rolling in. The first 6 are every 80 frames. The last 3 are every 40 frames.

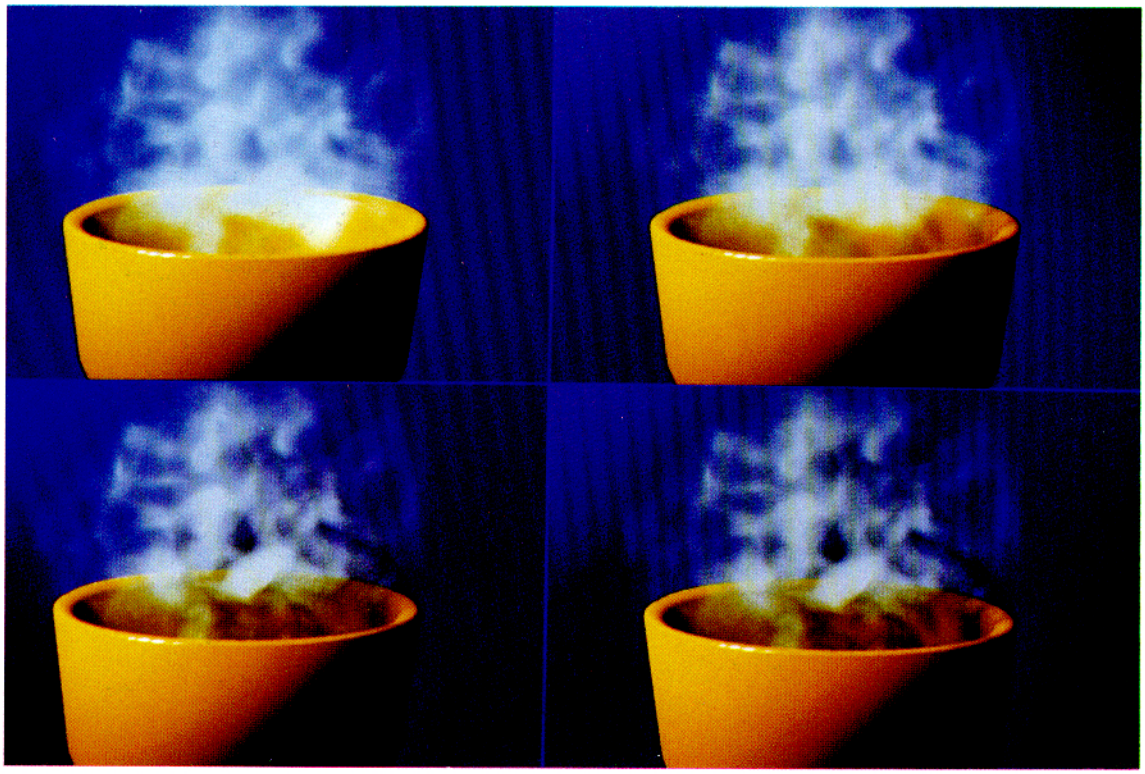

Figure 8: Steam rising from a glass. (a) has no shadowing (b) has shadowing of the gas onto the glass. (c) has table-based volume shadowing and self-shadowing. (d) has shadow-traced volume shadowing and self-shadowing.