Real-Time Cloud Rendering

Mark J. Harris and Anselmo Lastra

Department of Computer Science, University of North Carolina, Chapel Hill, North Carolina, USA

{harrism, lastra}@cs.unc.edu

Abstract

This paper presents a method for realistic real-time rendering of clouds suitable for flight simulation and games. It provides a cloud shading algorithm that approximates multiple forward scattering in a preprocess, and first order anisotropic scattering at runtime. Impostors are used to accelerate cloud rendering by exploiting frame-to-frame coherence in an interactive flight simulation. Impostors are shown to be particularly well suited to clouds, even in circumstances under which they cannot be applied to the rendering of polygonal geometry. The method allows hundreds of clouds and hundreds of thousands of particles to be rendered at high frame rates, and improves interaction with clouds by reducing artifacts introduced by direct particle rendering techniques.

1. Introduction

"Beautiful day, isn't it?"
"Yep. Not a cloud in the sky!"

Uncommon occurrences such as cloudless skies often elicit such common figures of speech. Clouds are such an integral feature of our skies that their absence from any synthetic outdoor scene can detract from its realism. Unfortunately, outdoor scenes in interactive applications such as flight simulators often suffer from cloudless skies. Designers of these applications have relied upon similar techniques to those used by renaissance painters in ceiling frescos: distant and high-flying clouds are represented by paintings on an always-distant sky dome. In addition, clouds in flight simulators and games have been hinted at with planar textures – both static and animated – or with semi-transparent textured objects and fogging effects.

There are many desirable effects associated with clouds that are not achievable with such techniques. In an interactive flight simulation, we would like to fly in and around realistic, volumetric clouds, and to see other flying objects pass within and behind them. Current real-time techniques have not provided users with such experiences. This paper describes a system for real-time cloud rendering that is appropriate for games and flight simulators.

In this paper we focus on high-speed, high-quality rendering of constant-shape clouds for games and flight simulators. These systems are already computationally and graphically loaded, so cloud rendering must be very fast. For this reason, we render realistically shaded static clouds, and do not address issues of dynamic cloud simulation. This choice enables us to generate clouds ahead of time, and to assume that cloud particles are static relative to each other. This assumption speeds the rendering of the clouds because we need only shade them once per scene in a preprocess.

Figure 1: *Realistic clouds in the game "Ozzy's Black Skies".*

The rest of this section presents previous work. Section 2 gives a derivation and description of our shading algorithm. Section 3 discusses dynamically generated impostors and shows how we use them to accelerate cloud rendering. We also discuss how we have dealt with issues in interacting with clouds. Section 4 discusses our results and presents performance measurements. We conclude and discuss ideas for future research in section 5.

Figure 2: *A view from an interactive flight through clouds.*

1.1 Previous Work

We segment previous work related to cloud rendering into two areas: cloud modeling and cloud rendering. Cloud modeling deals with the data used to represent clouds in the computer, and how the data are generated and organized. We build our clouds with particle systems. Reeves introduced particle systems as an approach to modeling clouds and other such "fuzzy" phenomena in [Reeves1983]. Voxels are another common representation for clouds. Voxel models provide a uniform sampling of the volume, and can be rendered with both forward and backward methods. Procedural solid noise techniques are also important to cloud modeling as a way to generate random but continuous density data to fill cloud volumes [Lewis1989, Perlin1985, Ebert1998].

Much previous work has been done in non-interactive rendering techniques for clouds. Rendering clouds is difficult because realistic shading requires the integration of the effects of optical properties along paths through the cloud volume, while incorporating the complex scattering within the medium. Previous work has attempted to approximate the physical characteristics of clouds at various levels of accuracy and complexity, and then to use these approximate models to render images of clouds. Blinn introduced the use of density models for image synthesis in [Blinn1982], where he presented a low albedo, single scattering approximation for illumination in a uniform medium. Kajiya and Von Herzen extended this work with methods to ray trace volume data exhibiting both single and multiple scattering [Kajiya1984]. Nelson Max provided an excellent survey in which he summarized the spectrum of optical models used in volume rendering and derived their integral equations from physical models [Max1995]. David Ebert has done much work in modeling "solid spaces", including offline computation of realistic images of smoke, steam, and clouds [Ebert1990, Ebert1997]. Nishita et al. introduced approximations and rendering techniques for global illumination of clouds accounting for multiple anisotropic scattering and skylight [Nishita1996].

Our rendering approach draws most directly from recent work by Dobashi *et al*,. which presents both an efficient simulation method for clouds and a hardware-accelerated rendering technique [Dobashi2000]. The shading method presented by Dobashi et al. implements an isotropic single scattering approximation. We extend this method with an approximation to multiple forward scattering and anisotropic first order scattering. The animated cloud scenes of Dobashi *et al.* required 20-30 seconds rendering time per frame. Our system renders static cloudy scenes at tens to hundreds of frames per second, depending on scene complexity.

2. Shading and Rendering

Particle systems are a simple and efficient method for representing and rendering clouds. Our cloud model assumes that a particle represents a roughly spherical volume in which a Gaussian distribution governs the density falloff from the center of the particle. Each particle is made up of a center, radius, density, and color. We get good approximations of real clouds by filling space with particles of varying size and density. Clouds in our system can be built by filling a volume with particles, or by using an editing application that allows a user to place particles and build clouds interactively. The randomized method is a good way to get a quick field of clouds, but we intend our clouds for interactive games with levels designed and built by artists. Providing an artist with an editor allows the artist to produce beautiful clouds tailored to the needs of the game.

We render particles using splatting [Westover1991], by drawing screen-oriented polygons texture-mapped with a Gaussian density function. Although we choose a particle system representation for our clouds, it is important to note that both our shading algorithm and our fast rendering system are independent of the cloud representation, and can be used with any model composed of discrete density samples in space.

2.1 Light Scattering Illumination

Scattering illumination models simulate the emission and absorption of light by a medium as well as scattering through the medium. *Single scattering* models simulate scattering through the medium in a single direction. This direction is usually the direction leading to the point of view. *Multiple scattering* models are more physically accurate, but must account for scattering in all directions (or a sampling of all directions), and therefore are much more complicated and expensive to evaluate. The rendering algorithm presented by Dobashi *et al*. computes an approximation of illumination of clouds with single scattering. This approximation has been used previously to render clouds and other participating media [Blinn1982, Kajiya1984].

In a multiple scattering simulation that samples N directions on the sphere, each additional order of scattering that is simulated multiplies the number of simulated paths by N. Fortunately, as demonstrated by [Nishita1996], the contribution of most of these paths is insignificant. Nishita

et al. found that scattering illumination is dominated by the first and second orders, and therefore they only simulated up to the 4th order. They reduce the directions sampled in their evaluation of scattering to sub-spaces of high contribution, which are composed mostly of directions near the direction of forward scattering and those directed at the viewer. We simplify further, and approximate multiple scattering only in the light direction – or *multiple forward scattering* – and anisotropic single scattering in the eye direction.

Our cloud rendering method is a two-pass algorithm similar to the one presented in [Dobashi2000]: we precompute cloud shading in the first pass, and use this shading to render the clouds in the second pass. The algorithm of Dobashi *et al.*, however, uses only an isotropic first order scattering approximation. If realistic values are used for the optical depth and albedo of clouds shaded with only a first order scattering approximation, the clouds appear very dark [Max1995]. This is because much of the illumination in a cloud is a result of light scattered forward along the light direction. Figures 8 and 9 show the difference in appearance between clouds shaded with and without our multiple forward scattering approximation.

2.1.1 Multiple Forward Scattering

The first pass of our shading algorithm computes the amount of light *incident* on each particle P in the light direction, l. This light, $I(P, l)$, is composed of all direct light from direction l that is not absorbed by intervening particles, plus light scattered to P from other particles. The multiple scattering model is written

$$I(P,\omega) = I_0 \cdot e^{-\int_0^{D_P} \tau(t)dt} + \int_0^{D_P} g(s,\omega) e^{-\int_s^{D_P} \tau(t)dt} ds, \quad (1)$$

where D_P is the depth of particle P in the cloud along the light direction, and

$$g(x,\omega) = \int_{4\pi} r(x,\omega,\omega') I(x,\omega') d\omega' \quad (2)$$

represents the light from all directions ω' scattered into direction ω at the point x. Here $r(x,\omega,\omega')$ is the bi-directional scattering distribution function (BSDF), and determines the percentage of light incident on x from direction ω' that is scattered in direction ω. It expands to $r(x,\omega,\omega') = a(x) \cdot \tau(x) \cdot p(\omega,\omega')$, where $\tau(x)$ is the optical depth, $a(x)$ is the albedo, and $p(\omega,\omega')$ is the phase function.

A full multiple scattering algorithm must compute this quantity for a sampling of all light flow directions. We simplify our approximation to compute only multiple forward scattering in the light direction, so $\omega = l$, and $\omega' = -l$. Thus, (2) reduces to $g(x,l) = r(x,l,-l) \cdot I(x,-l) / 4\pi$.

We split the light path from 0 to D_P into discrete segments s_j, for j from 1 to N, where N is the number of cloud particles along the light direction from 0 to D_P. By approximating the integrals with Riemann Sums, we have

$$I_P = I_0 \cdot \prod_{j=1}^{N} e^{-\tau_j} + \sum_{j=1}^{N} g_k \prod_{k=j+1}^{N} e^{-\tau_k}. \quad (3)$$

I_0 is the intensity of light incident on the edge of the cloud. In discrete form $g(x,l)$ becomes $g_k = a_k \cdot \tau_k \cdot p(l,-l) \cdot I_k / 4\pi$. We assume that albedo and optical depth are represented at discrete samples (particles) along the path of light. In order to easily transform (3) into an algorithm that can be implemented in graphics hardware, we cast it as a recurrence relation:

$$I_k = \begin{cases} g_{k-1} + T_{k-1} \cdot I_{k-1}, & 2 \le k \le N \\ I_0, & k = 1 \end{cases}. \quad (4)$$

If we let $T_k = e^{-\tau_k}$ be the transparency of particle p_k, then (4) expands to (3). This representation can be more intuitively understood. It simply says that starting outside the cloud, as we trace along the light direction the light incident on any particle p_k is equal to the intensity of light scattered to p_k from p_{k-1} plus the intensity transmitted through p_{k-1} (as determined by its transparency, T_{k-1}). Notice that if g_k is expanded in (4) then I_{k-1} is a factor in both terms. Section 2.3.1 explains how we combine frame buffer read back with hardware blending to evaluate this recurrence.

2.1.2 Eye Scattering

In addition to our multiple forward scattering approximation, which we compute in a pre-process, we also implement single scattering toward the viewer as in [Dobashi2000]. The recurrence for this is subtly different:

$$E_k = S_k + T_k \cdot E_{k-1}, \qquad 1 \le k \le N. \quad (5)$$

This says that the light, E_k, exiting any particle p_k is equal to the light incident on it that it does not absorb, $T_k \cdot E_{k-1}$, plus the light that it scatters, S_k. In the first pass, we were computing the light I_k incident on each particle from the light source. Now, we are interested in the portion of this light that is scattered toward the viewer. When S_k is replaced by $a_k \cdot \tau_k \cdot p(\omega, -l) \cdot I_k / 4\pi$, where ω is the view direction and T_k is as above, this recurrence approximates single scattering toward the viewer. It is important to mention that (5) computes light emitted from particles using results (I_k) computed in (4). Since illumination is multiplied by the phase function in both recurrences, one might think that the phase function is multiplied twice for the same light. This is not the case, since in (4), I_{k-1} is multiplied by the phase function to determine the amount of light P_{k-1} scatters to P_k in the light direction, and in (5) I_k is multiplied by the phase function to determine the amount of light that P_k scatters in the view direction. Even if the viewpoint is directly opposite the light source, since the light *incident* on P_k is stored and

used in the scattering computation, the phase function is never taken into account twice at the same particle.

2.1.3 Phase Function

The phase function, $p(\omega,\omega')$ mentioned above is very important to cloud shading. Clouds exhibit anisotropic scattering of light (including the strong forward scattering that we assume in our multiple forward scattering approximation). The phase function determines the distribution of scattering for a given incident light direction. Phase functions are discussed in detail in [Nishita1996], [Max1995], and [Blinn1982], among others. The images shown in this paper were generated using a simple Rayleigh scattering phase function, $p(\theta) = 3/4(1 + \cos^2\theta)$, where θ is the angle between the incident and scattered directions. Rayleigh scattering favors scattering in the forward and backward directions. Figures 10 and 11 demonstrate the differences between clouds shaded with and without anisotropic scattering. Anisotropic scattering gives the clouds their characteristic "silver lining" when viewed looking into the sun.

2.1 Rendering Algorithm

Armed with recurrences (4) and (5) and a standard graphics API such as OpenGL or Direct3D, computation of cloud illumination is straightforward. Our algorithm is similar to the one presented by [Dobashi2000] and has two phases: a shading phase that runs once per scene and a rendering phase that runs in real time. The key to the implementation is the use of hardware blending and pixel read back.

Blending operates by computing a weighted average of the frame buffer contents (the *destination*) and an incoming fragment (the *source*), and storing the result back in the frame buffer. This weighted average can be written

$$c_{result} = f_{src} \cdot c_{src} + f_{dest} \cdot c_{dest} \quad (6)$$

If we let $c_{result} = I_k$, $f_{src} = 1$, $c_{src} = g_{k-1}$, $f_{dest} = T_{k-1}$, and $c_{dest} = I_{k-1}$, then we see that (4) and (6) are equivalent if the contents of the frame buffer before blending represent I_0. This is not quite enough, though, since as we saw before, I_{k-1} is a factor of both terms in (4). To solve the recurrence for a particle p_k, we must know how much light is incident on particle p_{k-1} beforehand. To do this, we employ pixel read back.

To compute (4) and (5), we use the procedure described by the pseudocode in figure 3. The pseudocode shows that we use a nearly identical algorithm for preprocess and runtime. The differences are as follows. In the illumination pass, the frame buffer is cleared to white and particles are sorted with respect to the light. As a particle is blended into the frame buffer, the transparency of the particle modulates the color and adds an amount proportional to the forward scattering. The percentage of light that reaches p_k, is found by reading back the color of the pixel in the frame buffer to which the center of the particle projects immediately before rendering p_k.. I_k is computed by multiplying this percentage by the light intensity. I_k is used to compute multiple forward scattering in (4) and eye scattering in (5).

In the runtime phase we use the same algorithm, but with particles sorted with respect to the viewpoint, and without reading pixels. The precomputed illumination of each particle I_k is used in this phase to compute scattering toward the eye.

In both passes, particles are rendered in sorted order as polygons textured with a Gaussian function. The polygon

```
Source_blend_factor = 1;
dest_blend_factor = 1 – src_alpha;
texture mode = modulate;
l = direction from light;
if (preprocess) then
  ω = -l;
  view cloud from light source;
  clear frame buffer to white;
  particles.Sort(<, distance to
light);
else
  view cloud from eye position;
  particles.Sort(>,distance from
eye);
endif
  [Sort(<, distance from x) means
  sort in ascending order by distance
  from x, and > means sort in
  descending order]
foreach particle p_k
  [p_k has extinction τ_k, albedo a_k,
  radius r_k, color, and alpha]
  if (preprocess) then
    x = pixel at proj. center of p_k;
    i_k = color(x) * light_color;
    p_k.color = a_k * τ_k * i_k / 4π;
    p_k.alpha = 1 - exp(-τ_k);
  else
    ω = p_k.position – view_position;
  endif
  c = p_k.color * phase(ω, l);
  render p_k with color c, side 2*r_k;
end
```

Figure 3: *Pseudocode for cloud shading and rendering.*

color is set to the scattering factor $a_k \cdot \tau_k \cdot p(\omega,l) \cdot I_k / 4\pi$ and the texture is modulated by this color. In the first pass, ω is the light direction, and in the second pass it is the direction of the viewer. The source and destination blending factors are set to 1 and one minus source alpha, respectively. All cloud images in this paper and the accompanying video were computed with a constant τ of 8.0, and an albedo of 0.9.

2.2.1 Skylight

The most awe-inspiring images of clouds are provided by the multi-colored spectacle of a beautiful sunrise or sunset. These clouds are often not illuminated directly by the sun at all, but by skylight – sunlight that is scattered by the atmosphere. The fact that light accumulates in an additive manner provides us with a simple extension to our shading method that allows the creation of such beautiful clouds. We simply shade clouds from multiple light sources and store the resulting particle colors (i_k in the algorithm above) from all shading iterations. At render time, we evaluate the phase function at each particle once per light. By doing so, we can approximate global illumination of the clouds.

While this technique is not completely physically-based, it is better than an ambient contribution, since it is directional and results in shadowing in the clouds as well as anisotropic scattering from multiple light directions and intensities. We find that best results are obtained by guiding the placement and color of these lights using the images that make up the sky dome we place in the distance over our environments. Figure 12 demonstrates this with a scene at sunset in which we use two light sources, one orange and one pink, to create sunset lighting. In addition to illumination from multiple light sources, we provide an ambient term to provide some compensation for lost scattered light due to our scattering approximation.

3. Dynamically Generated Impostors

While the cloud rendering method described above provides beautiful results and is fast for relatively simple scenes, it suffers under the weight of many complex clouds. The interactive games for which we developed this system dictate that we must render complicated cloud scenes at fast interactive rates. Clouds are only one component of a complex game environment, and therefore can only use a small percentage of a frame time, with frame rates of thirty per second or higher. With direct particle rendering, even a scene with a few tens of thousands of particles is prohibitively slow on current hardware.

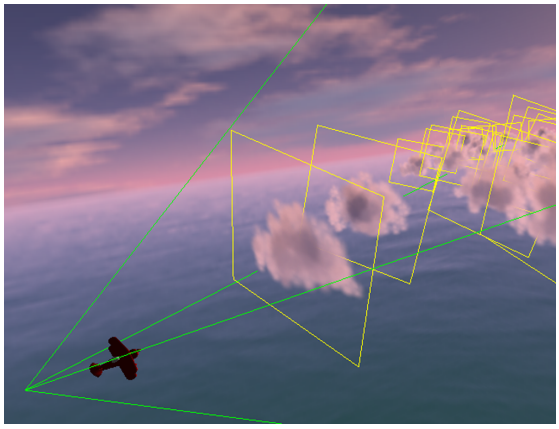

Figure 4: *Impostors, outlined here, are textured polygons oriented toward the viewer.*

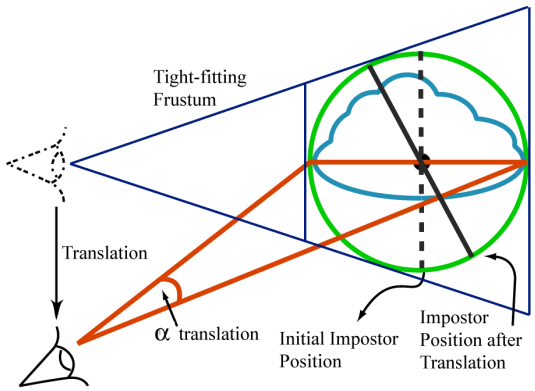

Figure 5: *Impostor generation and translation error metric.*

The integration required to accurately render volumetric media results in high rates of pixel overdraw. Clouds have inherently high depth complexity, and require blending, making rendering them a difficult job even for current hardware with the highest fill rates. In addition, as the viewpoint approaches a cloud, the projected area of that cloud's particles increases, becoming greatest when the viewpoint is within the cloud. Thus, pixel overdraw is increased and rendering slows as the viewpoint nears and enters clouds.

In order to render many clouds made up of many particles at high frame rates, we need a way to bypass fill rate limitations, either by reducing the amount of pixel overdraw performed, or by amortizing the rendering of cloud particles over multiple frames. *Dynamically generated impostors* allow us to do both.

[Maciel1995], [Schaufler1995], and [Shade1996] all discuss impostors. An impostor replaces an object in the scene with a semi-transparent polygon textured-mapped with an image of the object it replaces (figure 4). The image is a rendering of the object from a viewpoint V that is valid (within some error tolerance) for viewpoints near V. Impostors used for appropriate points of view give a very close approximation to rendering the object itself. An impostor is valid (with no error) for the viewpoint from which its image was generated, regardless of changes in the viewing direction. Impostors may be precomputed for an object from multiple viewpoints, requiring much storage, or they may be generated only when needed. We choose the latter technique, called *dynamically generated impostors* by [Schaufler1995].

We generate impostors using the following procedure. A view frustum is positioned so that its viewpoint is at the position from which the impostor will be viewed, and it is tightly fit to the bounding volume of the object (figure 5). We then render the object into an image used to texture the impostor polygon.

As mentioned above, we can use impostors to amortize the cost of rendering clouds over multiple frames. We do this by exploiting the frame-to-frame coherence inherent in three-dimensional scenes: the relative motion of objects in a scene decreases with distance from the viewpoint, and objects close to the viewpoint present a similar image for some time. This lack of sudden changes in the image of an object allows us to re-use impostor images over multiple frames. We can compute an estimate of the error in an impostor representation that we use to determine when the impostor needs to be updated. Certain types of motion introduce error in impostors more quickly than others. [Schaufler1995] presents two worst-case error metrics for this purpose. The first, which we will call the *translation error*, computes error caused by translation from the viewpoint at which the current impostor was generated. The second computes error introduced by moving straight toward the object, which we call the *zoom error*.

We use the same translation error metric, and replace zoom error by a texture resolution error metric. For the translation error metric, we simply compute the angle α_{trans}, shown in figure 5, and compare it to a specified tolerance. The zoom error metric compares the current impostor texture resolution to the required resolution for the texture, computed using the following equation [Schaufler1995]

$$resolution_{texture} = resolution_{screen} \cdot \frac{object\ size}{object\ dist}.$$

If either the translation error is greater than an error tolerance angle or the current resolution of the impostor is less than the required resolution, we regenerate the impostor from the current viewpoint. We find that a tolerance of about 0.15 degree reduces impostor "popping" to an imperceptible level while maintaining good performance. For added performance, tolerances up to one degree can be used without excessive popping.

In the past, impostors were used mostly to replace geometric models. Since these models have high frequencies in the form of sharp edges, impostors have usually been used only for distant objects. Nearby objects must have impostor textures of a resolution at or near that of the screen, and their impostors require frequent updates. We use impostors for clouds no matter where they are in relation to the viewer. Clouds do not have high frequency edges like those of geometric models, so artifacts caused by low texture resolution are less noticeable. Clouds have very high fill rate requirements, so cloud impostors are beneficial even when they must be updated every few frames.

Figure 6: *An airplane in the clouds. On the left, particles are directly rendered into the scene. Artifacts of their intersection with the plane are visible. On the right, the airplane is rendered between impostor layers, and no artifacts are visible.*

3.1 Head in the Clouds

Impostors can provide a large reduction in overdraw even for viewpoints inside the cloud, where the impostor must be updated every frame. The "foggy" nature of clouds makes it difficult for the viewer to discern detail when inside them. In addition, in games and flight simulators, the viewpoint is often moving. These factors allow us to reduce the resolution at which we render impostor textures for clouds containing the viewpoint by about a factor of 4 in each dimension.

However, impostors cannot be generated in the same manner for these clouds as for distant clouds, since the view frustum cannot be tightly fit to the bounding volume as described above. Instead, we use the same frustum used to display the whole scene to generate the texture for the impostor, but create the texture at a lower resolution, as described above. We display these impostors as screen-space rectangles sized to fill the screen.

3.1.1 Objects in the Clouds

In order to create effective interactive cloudy scenes, we must allow objects to pass in and through the clouds, and we must render this realistically. Impostors pose a problem because they are two-dimensional. Objects that pass through impostors appear as if they are passing through images floating in space, rather than through fluffy, volume-filling clouds.

One way to solve this problem would be to detect clouds that contain objects and render their particles directly to the frame buffer. By doing so, however, we lose the benefits that impostors provide us. Instead, we detect when objects pass within the bounding volume of a cloud, and split the impostor representing that cloud into multiple layers. If only one object resides in a certain cloud, then that cloud is rendered as two layers: one for the portion of cloud particles that lies approximately behind the object with respect to the viewpoint, and one for the portion that lies approximately in front of the object. If two objects lie within a cloud, then we need three layers, and so on. Since cloud particles must be

sorted for rendering anyway, splitting the cloud into layers adds little expense. This "impostor splitting" results in a set of alternating impostor layers and objects. This set is rendered from back to front, with depth testing enabled for objects, and disabled for impostors. The result is an image of a cloud that realistically contains objects, as shown on the right side of figure 6.

Impostor splitting provides an additional advantage over direct particle rendering for clouds that contain objects. When rendering cloud particles directly, the billboards used to render particles may intersect the geometry of nearby objects. These intersections cause artifacts that break the illusion of particles representing elements of volume. Impostor splitting avoids these artifacts (figure 6).

4. Results and Discussion

We have implemented our cloud rendering system using both the OpenGL and DirectX 8 APIs. We have tested the OpenGL-based system on both Windows PC systems and an SGI Onyx2 with InfiniteReality2 graphics. On a PC with an NVIDIA GeForce graphics processor, we can achieve very high frame rates by using impostors and view-frustum culling to accelerate rendering. We can render scenes containing up to hundreds of thousands of particles at high frame rates (greater than 50 frames per second). If the viewpoint moves slowly enough to keep impostor update rates low, we can render a scene of more than 1.2 million particles at about 10 to 12 frames per second. Slow movement is a reasonable assumption for flight simulators and games because the user's aircraft is typically much smaller than the clouds through which it is flying, so the frequency of impostor updates remains low.

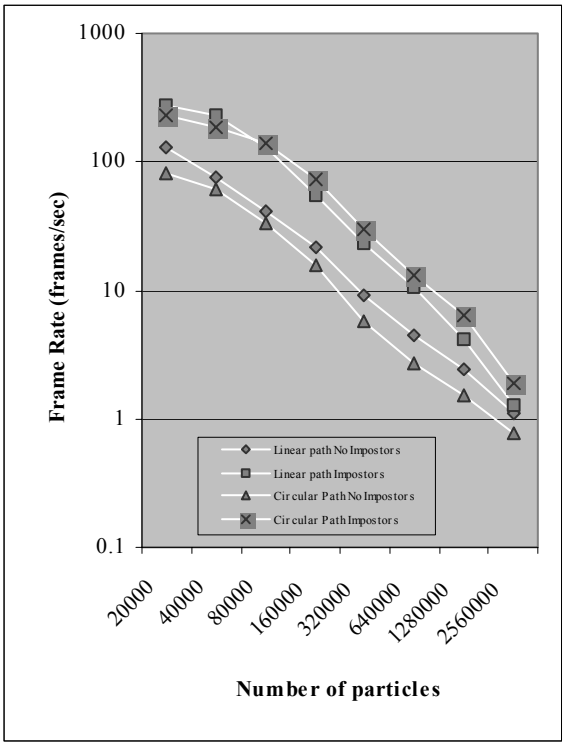

Figure 7: *Results of performance measurements for cloudy scenes of varying complexity rendered with and without impostors.*

As mentioned before, our shading phase is a preprocess. For scenes with only a few thousand particles shading takes less than a second, and scenes of a few hundreds thousand particles can be shaded in about five to ten seconds per light source.

We have performed several tests of our cloud system and present the results here. Our test machine is a PC with 256 MB of RAM and an Intel Pentium III processor running at 800 MHz. It uses an NVIDIA GeForce 256 graphics card with 32MB of video RAM.

The tests rendered scenes of increasing cloud complexity (from 100 to 12800 clouds of 200 particles each) with and without using impostors. We also tested the performance for different types of movement. The first test moved the camera around a circular path, and the second moved the camera through the clouds in the direction of view. The results of our tests are shown in figure 7. The chart shows that using impostors was faster than not using them for the large range of scene complexity that we covered, and that even for scenes with several hundred thousand particles we achieve interactive frame rates.

Our cloud rendering algorithms have been incorporated into the game "Ozzy's Black Skies", by iROCK Interactive. In this game, players ride fantastical flying creatures through beautiful environments with realistically shaded volumetric clouds. The clouds are interesting in an interactive sense, as players may momentarily hide in them as they pass through. The steps we have taken to ensure high frame rates for cloud rendering makes our system work well in an already graphics- and computation-laden game engine. Impostors provide a means of scalability that is necessary in games intended to run on a wide range of hardware. Performance and quality can be balanced by adjusting impostor error tolerances and texture resolution.

5. Conclusion and Future Work

We have presented methods for shading and rendering realistic clouds at high frame rates. Our shading and rendering algorithm simulates multiple scattering in the light direction, and anisotropic single scattering in the view direction. Clouds are illuminated by multiple directional light sources, with anisotropic scattering from each.

Our method uses impostors to accelerate cloud rendering by exploiting frame-to-frame coherence and greatly reducing pixel overdraw. We have shown that impostors are an advantageous representation for clouds even in situations where they would not be successfully used to represent other objects, such as when the viewpoint is in or near a cloud. Impostor splitting is an effective way to render clouds that

contain other objects, reducing artifacts caused by direct particle rendering.

Since our shading algorithm computes multiple forward scattering during the illumination phase, it should be straightforward to extend it to compute an approximation of global multiple scattering. This would require running many passes to evenly sample all directions, and accumulating the results at the particles. We are also researching methods for speeding cloud shading by avoiding pixel read back, so that we can shade and render dynamic clouds in real time. This will allow the visualization of cloud formation in an interactive simulation.

Currently we are limited in the size of clouds that we are able to render at high frame rates, because large clouds require high-resolution impostors that are expensive to update. We would like to render dense fields of immense cumulonimbus clouds in real time. In order to solve this problem, we will explore hierarchical image caching [Shade1996] and other work that has been done with impostors.

Beyond clouds, we think that other phenomena might benefit from our shading algorithm. For example, we would like to be able to render realistic interactive flight through stellar nebulae. We have ideas for representing nebulae as particle clouds with emissive properties, and rendering them with a modified version of our algorithm.

Acknowledgements

This work would not have been possible without the support, encouragement, and ideas of the developers at iROCK Interactive, especially Wesley Hunt, Paul Rowan, Brian Stone, and Robert Stevenson. Mary Whitton and Frederick Brooks at UNC provided help and support. Rui Bastos gave help with the paper and ideas for the future, and Sharif Razzaque helped with modeling. This work was supported by NIH National Center for Research Resources, Grant Number P41 RR 02170 and iROCK Interactive.

References

[Blinn1982] J. Blinn, "Light Reflection Functions for Simulation of Clouds and Dusty Surfaces". *SIGGRAPH* 1982, pp. 21-29

[Dobashi2000] Y. Dobashi, K. Kaneda, H. Yamashita, T. Okita, and T. Nishita. "A Simple, Efficient Method for Realistic Animation of Clouds". *SIGGRAPH* 2000, pp. 19-28

[Ebert1990] D. S. Ebert, R. E. Parent, "Rendering and Animation of Gaseous Phenomena by Combining Fast Volume Scanline A-Buffer Techniques," *SIGGRAPH* 1990, pp. 357-366.

[Ebert1997] D. S. Ebert, "Volumetric Modeling with Implicit Functions: A Cloud is Born," *Visual Proceedings of SIGGRAPH* 1997, pp.147.

[Ebert1998] D. S. Ebert, F.K. Musgrave, D. Peachey, K. *Perlin*, S. Worley, *Texturing & Modeling: a Procedural Approach*. 1998, AP Professional.

[Kajiya1984] J. Kajiya and B. Von Herzen. "Ray Tracing Volume Densities". *SIGGRAPH* 1984, pp. 165-174.

[Lewis1989] J. Lewis. "Algorithms for Solid Noise Synthesis". SIGGRAPH 1989, pp. 263-270.

[Maciel1995] P. Maciel, P. Shirley. "Visual Navigation of Large Environments Using Textured Clusters". *Proceedings of the 1995 symposium on Interactive 3D graphics*, 1995, Page 95

[Max1995] N. Max. "Optical Models for Direct Volume Rendering", IEEE *Transactions on Visualization and Computer Graphics*, vol. 1 no. 2, June 1995.

[Nishita1996] T. Nishita, Y. Dobashi, E. Nakamae. "Display of Clouds Taking into Account Multiple Anisotropic Scattering and Sky Light." *SIGGRAPH* 1996, pp. 379-386.

[Perlin1985] K. Perlin. An Image Synthesizer. *SIGGRAPH* 1985, pp. 287-296.

[Reeves1983] W. Reeves. "Particle Systems – A Technique for Modeling a Class of Fuzzy Objects". *ACM Transactions on Graphics*, Vol. 2, No. 2. April 1983. ACM.

[Reeves1985] W. Reeves and R. Blau. "Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particle Systems". SIGGRAPH 1985, pp. 313-322.

[Schaufler1995] G. Schaufler, "Dynamically Generated Impostors", GI Workshop *Modeling - Virtual Worlds - Distributed Graphics*, 1995, pp 129-136.

[Shade1996] J. Shade, D. Lischinski, D. Salesin, T. DeRose, J. Snyder, "Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments". *SIGGRAPH* 1996, pp. 75-82.

[Westover1990] L. Westover, "Footprint evaluation for volume rendering" *SIGGRAPH* 1990, Pages 367 - 376

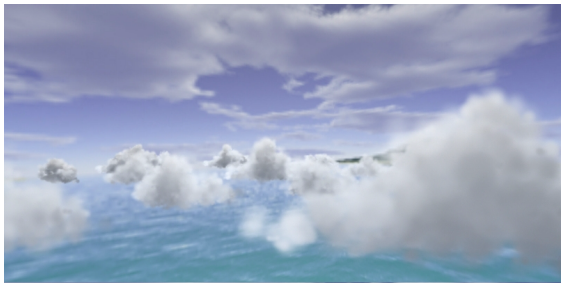

Figure 8: *Shading with multiple forward scattering.*

Figure 9*: Shading with only single scattering.*

Figure 10: *Clouds with anisotropic scattering.*

Figure 11: *Clouds with isotropic scattering.*

Figure 12: *An example of shading from two light sources to simulate sky light. This scene was rendered with two lights, one orange and one pink. Anisotropic scattering simulation accentuates the light coming from different directions. See section 2.*