Cloud Rendering for Interactive Applications

Mark J. Harris
Department of Computer Science, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA
harrism@cs.unc.edu

Figure 1: *Clouds add realism to interactive flight.*

1 INTRODUCTION

Clouds are an important feature of the sky; without them, synthetic outdoor scenes seem unrealistic. Game and flight simulator designers know this, so their applications nearly always have some form of clouds present. Applications in which the user's viewpoint stays near the ground can rely on techniques similar to those used by renaissance painters in ceiling frescos: distant and high-flying clouds are represented by paintings on an always distant sky dome. Flight simulators and other flying games don't have it so easy – their users are free to roam the sky.

Many techniques have been used for clouds in games and flight simulators. They have been represented with planar textures – both static and animated – or rendered as semi-transparent textured objects and fogging effects. These techniques leave a lot of effects to be desired. In a flight simulator, for example, we would like to fly in and around realistic, volumetric clouds, and to see other flying objects pass within and behind them. The goal of these course notes is to introduce the reader to existing techniques for modeling and rendering clouds. The emphasis is on techniques that are amenable to real-time implementation.

In particular, we will focus on high-speed, high-quality rendering of constant-shape clouds as described in [Harris and Lastra 2001]. We will concentrate on the rendering of realistically shaded static clouds, and will 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 cloud rendering because we need only shade them once per scene at application load time.

Before describing these techniques in detail, we provide background on other existing techniques, and on the basic physics underlying the interaction of light with clouds.

1.1 Previous Work

We will address two important areas of previous work: 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.

1.1.1 Cloud Modeling

As with the modeling of any object or phenomenon, there are multiple ways to represent clouds. The five techniques we will describe are particle systems, metaballs, voxel volumes, procedural noise, and textured solids. Note that these techniques are not mutually exclusive; elements of multiple techniques can be combined with good results.

Particle Systems

Particle systems model objects as a collection of *particles* – simple primitives that can be represented by a single 3D position and a small number of attributes such as radius, color, and texture. Reeves introduced particle systems as an approach to modeling clouds and other such "fuzzy" phenomena in [Reeves 1983], and described approximate methods of shading models composed of particles in [Reeves and Blau 1985]. Particles can be placed by hand using a modeling tool, procedurally generated, or with some combination of the two. Particles can be rendered in a variety of ways. The method we will describe in detail later in these notes builds clouds with particles, and renders each particle as a small, textured sprite (or "splat").

Particles have the advantage that they usually require only very simple and inexpensive code to maintain and render them. Because a particle implicitly represents a spherical volume, a cloud built with particles usually requires much less storage than a similarly detailed clouds represented with other methods. This advantage may diminish as detail increases, because many tiny particles are needed to achieve high detail. In such a situation other techniques, such as those described below, may be more desirable.

Figure 2: *These clouds, modeled with metaballs, exhibit multiple anisotropic scattering and are illuminated by sunlight and skylight. [Nishita, et al. 1996](Image courtesy of Tomoyuki Nishita)*

Metaballs

Metaballs (or "blobs") represent volumes as the superposition of potential fields represented as a set of sources defined by their center, radius, and strength [Blinn 1982a]. Such volumes can be rendered in a number of ways, including ray tracing and splatting. Alternatively, isosurfaces may be extracted and rendered (however this may not be appropriate for clouds). Metaballs were used for modeling clouds by [Nishita,

Figure 4: *This cloud was simulated using a Coupled Map Lattice model. [Miyazaki, et al. 2001](Image courtesy of Tomoyuki Nishita).*

Figure 4: *Multiple textured ellipsoids used to create clouds [Elinas and Stürzlinger 2001]. (Image courtesy of Wolfgang Stürzlinger.)*

et al. 1996], who first created a basic cloud shape by hand-placing a few metaballs, and then added detail via a fractal method of generating new metaballs on the surface of existing ones (Figure 2). Metaballs were used in [Dobashi, et al. 1999] to model clouds extracted from satellite images. In [Dobashi, et al. 2000], clouds in a voxel grid were converted into metaballs for rendering with splatting (Figure 6).

Voxel Volumes

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. There is a large body of existing work on volume rendering which can be drawn upon when rendering clouds represented as voxel volumes. [Kajiya and Von Herzen 1984] performed a simple physical simulation of clouds and stored the results in voxel volumes which they rendered using ray tracing.

As mentioned above, voxel grids are typically used when physically-based simulation is involved. [Dobashi, et al. 2000] simulated clouds on a voxel grid using a cellular automata model similar to [Nagel and Raschke 1992], and rendered them using metaballs, as mentioned above. [Miyazaki, et al. 2001] also performed cloud simulation on a grid using a method known as a Coupled Map Lattice (CML), and then rendered the resulting clouds in the same way. [Overby, et al. 2002] solved a set of partial differential equations to generate clouds on a voxel grid.

Procedural Noise

Procedural solid noise techniques are another important technique for generating models of clouds. These methods use noise as a basis, and perform various operations on the noise to generate random but continuous density data to fill cloud volumes [Lewis 1989;Perlin 1985]. David Ebert has done much work in modeling "solid spaces" using procedural solid noise, including offline computation of realistic images of smoke, steam, and clouds [Ebert 1997;Ebert and Parent 1990]. Figure 5 shows a cloud generated from a union of implicit functions. The

Figure 5: *A cloud generated using implicit functions and procedural noise. (Image courtesy of David Ebert.)*

solid space defined by the implicit functions is perturbed by procedural solid noise, and then rendered using a scan line renderer.

Textured Solids

Others have chosen to use surfaces to represent clouds rather than the volumetric methods described above. [Gardner 1985] used fractal texturing on the surface of ellipsoids to simulate the appearance of clouds. By combining multiple textured and shaded ellipsoids, he was able to create convincing cloudy scenes. [Lewis 1989] also demonstrated the use of ellipsoids for clouds, this time with procedural solid noise. More recently, [Elinas and Stürzlinger 2001] used a variation of Gardner's method to interactively render clouds composed of multiple ellipsoids (Figure 4).

1.1.2 Cloud Rendering

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 light scattering within the medium. Much effort has been made to approximate the physical characteristics of clouds at various levels of accuracy and complexity, and to use these approximate models to render images of clouds. Blinn introduced the use of density models for image synthesis in [Blinn 1982b], where he presented a low albedo, single scattering approximation for illumination in a uniform medium.

Kajiya and Von Herzen extended Blinn's work with methods to ray trace volume data exhibiting both single and multiple scattering [Kajiya and Von Herzen 1984]. Their method used two passes. In the first pass, scattering and absorption were integrated along paths from the light source through the cloud to each voxel where the resulting intensities were stored. In the second pass, eye rays were traced through the volume of intensities and scattering of light to the eye was computed, resulting in a cloud image. For multiple scattering, the authors derive a discrete spherical harmonic approximation to the multiple scattering equation, and solve the resulting matrix of partial differential equations using relaxation (This matrix solution replaces the first pass of the above algorithm). Following Kajiya and Von Herzen's lead, two pass techniques for computing light scattering in volumetric media – including the one we will present later – are now common.

Nishita et al. introduced additional approximations and rendering techniques for global illumination of clouds accounting for multiple anisotropic scattering and skylight [Nishita, et al. 1996]. In their method, illumination is computed on a voxel grid. The complexity of computing multiple scattering is high because it requires integrating illumination over the sphere of incoming directions. Nishita et al. showed how this complexity can be reduced by sampling only the most important directions on the sphere. Because scattering from cloud water droplets is anisotropic, with a strong peak in the forward direction, the number of sample directions is greatly reduced, saving a large amount of computation.

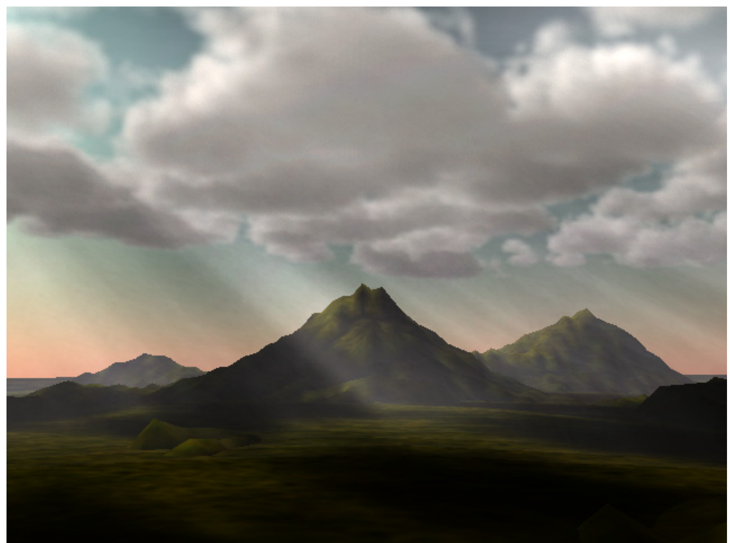


Figure 6: *These clouds were simulated using cellular automata and rendered using splatting. (Image courtesy of Tomoyuki Nishita.)*

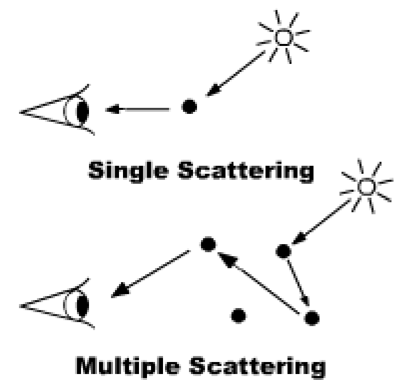
The rendering approach described in detail later in these notes draws most directly

from the rendering technique presented by [Dobashi, et al. 2000]. Their method renders clouds using a two-pass splatting algorithm in which the clouds are represented with particles. The first pass traverses the particles in sorted order moving away from the light source, using splatting and frame buffer read back to compute the amount of light that reaches each particle. In the second pass the particles are sorted with respect to the camera and then splatted from back to front into the frame buffer, using the illumination computed in the first pass as the particle color. The end result is a realistic, self-shadowing image of the cloud (Figure 6). The method we will describe later extends this method with an approximation to multiple anisotropic forward scattering. By computing the first pass only once at application load time, we are able to render static clouds at high frame rates.

2 Radiometry

This section provides a brief review of radiometric terminology. An excellent reference to the spectrum of optical models used in volume rendering, including derivations of the integral equations, is [Max 1995].

2.1 Essential Definitions



To improve clarity in the next few sections, we will review some basic radiometry terms. *Absorption* is the phenomenon by which light energy is converted into another form upon interacting with particles in a medium. For example, your skin warms in sunlight because some of the light is absorbed and transformed into heat energy. *Scattering* can be thought of as an elastic collision between matter and a photon in which the direction of the photon is changed. *Extinction*, K, describes the attenuation of light energy by absorption and scattering:

$$K = K_s + K_a, \tag{1}$$

where K_s is the coefficient of scattering and K_a is the coefficient of absorption. Any light that interacts with a medium undergoes either scattering or absorption. If it does not interact, then it is *transmitted*. Extinction (and therefore scattering and absorption) is proportional to density.

Single Scattering is scattering of light by a single particle. In optically thin media (media that are either physically very thin, or very transparent), scattering of light can be approximated using single scattering models. Clear air and steam from a cup of coffee can be approximated this way, but clouds cannot. *Multiple Scattering* is scattering of light from multiple particles in succession. Models that account for only single scattering cannot accurately represent optically thick media such as clouds. Multiple scattering is the reason that clouds appear much brighter (and whiter) than the sky, since most of the light that emerges from a cloud has been scattered many times.

The *Single Scattering Albedo* is the percentage of attenuation by extinction that is due to scattering, rather than absorption:

$$\varpi = \frac{\alpha}{\alpha + \beta}. \tag{2}$$

Single scattering albedo is the probability that a photon will “survive” an interaction with a medium. *Optical Depth* is a dimensionless measure of how opaque a medium is to light passing through it. It is the product of the physical material thickness, d, and the extinction coefficient K (assuming the material is homogeneous). An optical depth of 1 indicates that there is $e^{-1} \approx 37\%$ chance that the light will travel at least the distance d without scattering or absorbing. An optical depth of infinity means that the

medium is opaque. The exponential given above is the *transparency* of the medium. Thus a medium with optical depth of 1 is 37% transparent.

A *Phase Function* is a function that determines, for any angle between incident and outgoing directions (the *phase angle*), how much of the incident light intensity will be scattered in the outgoing direction. For example, scattering by very small particles such as those found in clear air can be approximated using *Rayleigh scattering* [Strutt 1871]. The phase function for Rayleigh scattering is

$$p(\theta) = \frac{3}{4}\left(1 + \cos^2\theta\right), \tag{3}$$

where θ is the phase angle. Gustav Mie developed a theory of scattering by larger particles [Mie 1908]. *Mie scattering* theory is much more complicated than Rayleigh scattering, but some simplifying assumptions can be made. A commonly used approximation for Mie scattering is the Henyey-Greenstein phase function [Henyey and Greenstein 1941]:

$$p_{HG}(\theta) = \frac{1}{4\pi}\frac{1-g^2}{(1-2g\cos\theta + g^2)^{3/2}}. \tag{4}$$

This is the polar form for an ellipse centered at one of its foci. Anisotropy of the scattering is controlled by g, the eccentricity of the ellipse. Positive values of g will cause most of the light to be scattered in the forward direction, negative values result in backward scattering, and $g = 0$ results in isotropic scattering.

2.2 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 in [Dobashi, et al. 2000] computes an approximation of cloud illumination with single scattering. This approximation has been used previously to render clouds and other participating media [Blinn 1982b;Kajiya and Von Herzen 1984].

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 [Nishita, et al. 1996], the contribution of most of these paths is insignificant to cloud rendering. 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

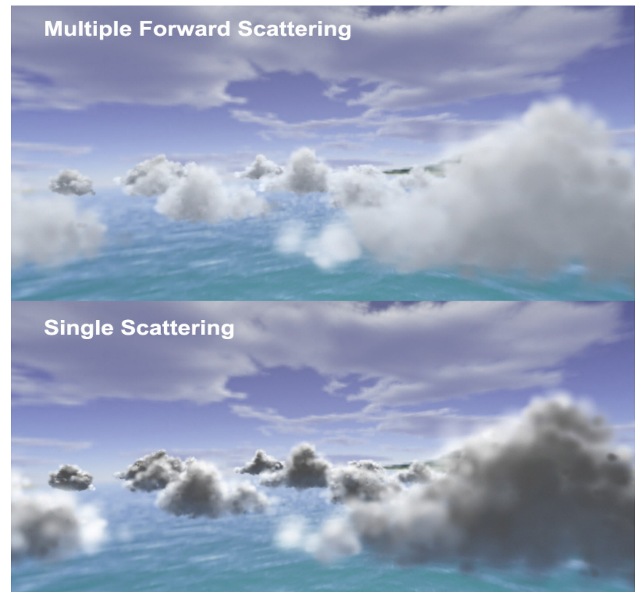


Figure 7: *A comparison of multiple forward scattering and single scattering approximations. Clouds shaded with only single scattering appear unrealistically dark.*

those directed at the viewer. Because of the dominance of the forward scattering direction, the technique we use simplifies even further, approximating 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 [Dobashi, et al. 2000]: 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 single scattering approximation. If realistic values are used for the optical depth and albedo of clouds shaded with only a single scattering approximation, the clouds appear very dark [Max 1995]. This is because much of the illumination in a cloud is a result of light scattered forward along the light direction. Figure 7 shows the difference in appearance between clouds shaded with and without the multiple forward scattering approximation.

2.2.1 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

$$I(P,\omega) = I_0 \cdot e^{-\int_0^{D_P} \tau(t)dt} + \int_0^{D_P} g(s,\omega) e^{-\int_s^{D_P} \tau(t)dt} ds, \tag{5}$$

where I_0 is the sunlight intensity incident on the cloud, D_P is the depth of particle P in the cloud along the light direction, and

$$g(x,\omega) = \int_{4\pi} r(x,\omega,\omega')I(x,\omega')d\omega' \tag{6}$$

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). It 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 $\tau(x)$ and $a(x)$ are the optical depth and scattering albedo at position x, and $p(\omega,\omega')$ is the phase function (explained later).

A full multiple scattering algorithm must compute this quantity for a sampling of all light flow directions. We simplify our approximation by only sampling a small solid angle around the forward light direction, and thus compute only multiple forward scattering. So, $\omega \approx l$, and $\omega' \approx -l$, and (6) reduces to $g(x,l) = r(x,l,-l)I(x,-l)/4\pi$.

We divide 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

$$I_P = I_0 \cdot \prod_{j=1}^{N} e^{-\tau_j} + \sum_{j=1}^{N} g_j \prod_{k=j+1}^{N} e^{-\tau_k}. \tag{7}$$

I_0 is the intensity of light incident on the edge of the cloud. In discrete form g(x,l) becomes $g_k = a_k \tau_k p(l,-l) I_k / 4\pi$, where intensity I_k, albedo a_k, and optical thickness τ_k are represented at discrete samples (the particles) along the path of light. In order to easily transform (7) into an algorithm that can be implemented in graphics hardware, we rewrite it as an equivalent recurrence relation:

$$I_k = \begin{cases} g_{k-1} + T_{k-1} \cdot I_{k-1}, & 2 \le k \le N \\ I_0, & k = 1 \end{cases} . \quad (8)$$

If we let $T_k = e^{-\tau_k}$ be the transparency of particle p_k, then (8) expands to (7). This representation can be intuitively understood. Starting outside the cloud, the intensity reaching particles at the cloud edge is I_0. As we trace into the cloud along the light direction, the light incident on particle k is equal to the intensity of light scattered to k from k-1 (the g_{k-1} term) plus the intensity transmitted through p_{k-1} (the $T_{k-1} \cdot I_{k-1}$ term). Notice that if g_{k-1} is expanded in (8) then I_{k-1} is a factor in both terms. Section 2.3 explains how we combine frame buffer read back with hardware blending to efficiently evaluate this recurrence.

2.2.2 Eye Scattering

In addition to multiple forward scattering and absorption, which we precompute, we also implement single scattering toward the viewer as in [Dobashi, et al. 2000]. The recurrence for this is subtly different:

$$E_k = S_k + T_k \cdot E_{k-1}, \qquad 1 \le k \le N. \quad (9)$$

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 described in the previous section, we computed the light I_k *incident* on each particle from the light source. In the second pass we are interested in the *exitant* portion of this light that is scattered toward the viewer. When S_k is replaced by $S_k = a_k \tau_k p(\omega, -l) I_k / 4\pi$, where ω is the view direction, and T_k is the same transparency factor used above, this recurrence approximates single scattering toward the viewer.

It is important to mention that (9) computes light emitted from particles using results (I_k) computed in (8). 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 (8), 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 (9) 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 when computing the *exitant* intensity.

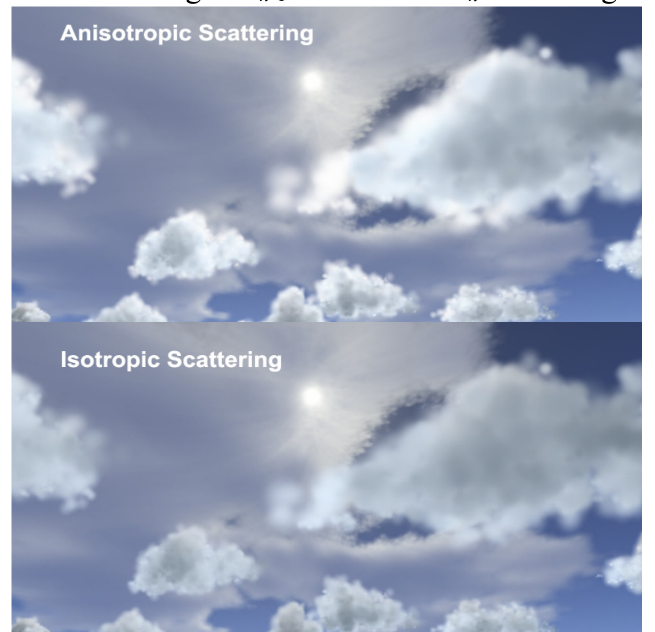


Figure 8: *A comparison of clouds rendered with isotropic and anisotropic scattering.*

2.2.3 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. The use of phase functions in cloud rendering is discussed in detail in [Blinn 1982b;Max 1995;Nishita, et al. 1996]. The

clouds in Figures 1, 7, 8, 10, 12, 13, and 14 were generated using a simple Rayleigh scattering phase function given in section 2.1. Rayleigh scattering favors scattering in the forward and backward directions. While a Mie scattering function would be more realistic, we have achieved good results with the simpler Rayleigh scattering model. Figure 8 demonstrates the differences between clouds shaded with and without anisotropic scattering. Anisotropic scattering gives the clouds a characteristic "silver lining" when viewed looking into the sun.

2.3 Rendering Algorithm

Armed with recurrences (8) and (9) and a standard graphics API such as OpenGL or Direct3D, computation of cloud illumination is straightforward. The algorithm described here is similar to the one presented by [Dobashi, et al. 2000] 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

$$c_{result} = f_{src} \cdot c_{src} + f_{dest} \cdot c_{dest} . \quad (10)$$

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 (8) and (10) 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 (8). 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 (8) and (9), we use the procedure described by the pseudocode in Figure 9. This 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, blending attenuates the intensity of each fragment by the opacity of the particle, and increases the intensity by the amount the particle scatters in the forward direction. The percentage of light that reaches p_k, is found by reading back the color of pixels in the frame buffer onto which the particle projects immediately before rendering it. I_k is computed by multiplying this percentage by

```
Source_blend_factor = 1;
destination_blend_factor = 1 – source_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(ascending order by distance to light);
}
else {
  view cloud from eye position;
  particles.Sort(descending order by distance to eye);
}
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 projected 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;
  }
  c = p_k.color * phase(ω, l);
  render p_k with color c, side 2*r_k;
}
```

Figure 9: *Pseudocode for illuminating and rendering clouds.*

the light intensity. I_k is used to compute multiple forward scattering in (8) and eye scattering in (9).

The runtime phase uses 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, we render particles in sorted order as polygons textured with a Gaussian "splat" texture. The polygon color is set to the scattering factor $S_k = a_k \tau_k p(\omega, -l) I_k / 4\pi$ 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 one and one minus source alpha, respectively. All cloud images in these notes were computed with a constant τ of 80.0 (units are $length^{-1}$), and an albedo of 0.9.

2.3.1 Skylight

The most awe-inspiring images of clouds are created 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 light approximation, since it is directional and results in shadowing in the clouds as well as anisotropic scattering from multiple light directions and intensities. We obtained best results by using the images that make up the sky dome we place in the distance over our environments to guide the placement and color of lights. Figure 14 shows 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 optionally use a small ambient term to provide some compensation for scattered light lost 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 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.

Figure 10: *Impostors, shown outlined in this image, are textured polygons oriented toward the viewer.*

The integration (section 2.2) 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 surmount 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.

Impostors are a common technique for accelerating interactive rendering [Maciel and Shirley 1995;Schaufler 1995;Shade, et al. 1996]. An impostor replaces an object in the scene with a semi-transparent polygon texture-mapped with an image of the object it replaces (Figure 10). 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 [Schaufler 1995].

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 11). We then render the object into an image used to texture the impostor polygon.

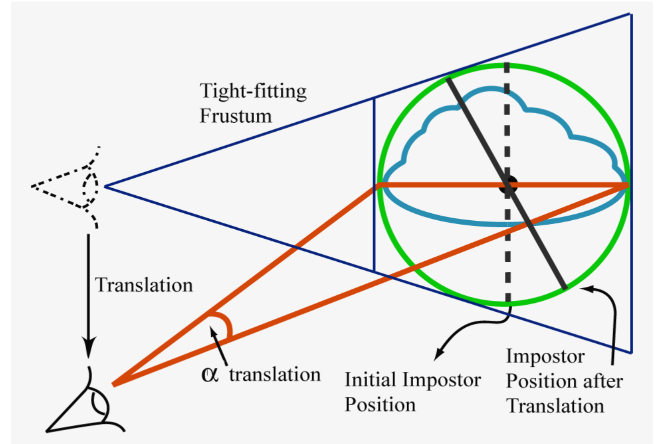


Figure 11: *Impostor translation error metric.*

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 [Schaufler 1995] presents two worst-case error metrics for this purpose. The first, which we will call the *translation error*, computes error caused by translation away 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 11, 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 [Schaufler 1995]

$$resolution_{texture} = resolution_{screen} \cdot \frac{object\ size}{object\ dist}. \qquad (11)$$

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 with more noticeable (but not 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. The couds we model have very few high frequency details 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.

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.

3.2 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. But by doing so we would sacrifice 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. When an object resides inside a cloud, the 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

Figure 12 *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.*

disabled for impostors. The result is an image of a cloud that realistically contains objects, as shown on the right side of Figure 12.

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 12).

Figure 13: *Clouds rendered in real time in* SkyWorks*.*

4 Results

We have implemented the cloud rendering system described here using the OpenGL API. The code was originally developed to run on a PC with an NVIDIA GeForce 256 graphics processor (circa 1999). Even on older graphics cards like this, we can achieve very high frame rates by using impostors and view-frustum culling to accelerate rendering. Scenes containing hundreds of thousands of particles render at 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. On the most recent hardware, performance is much higher. Much more complex scenes can be rendered at over 100 frames per second.

As mentioned before, cloud shading computations are performed in a preprocess. For scenes with only a few thousand particles shading takes less than a second and scenes of a few hundred thousand particles can be shaded in a few seconds per light source.

SkyWorks, an efficient open source implementation of this cloud rendering system, can be downloaded at http://www.cs.unc.edu/~harrism/SkyWorks (Figure 13).

5 Conclusion

These notes presented methods for shading and rendering realistic clouds at high frame rates. The shading and rendering algorithm simulates multiple scattering in the light direction, and anisotropic single scattering in the view direction. Clouds can be illuminated by multiple directional light sources, with anisotropic scattering from each.

This method uses impostors to accelerate cloud rendering by exploiting frame-to-frame coherence and greatly reducing pixel overdraw. 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.

6 Acknowledgements

This work was supported by iROCK Interactive, NVIDIA Corporation, NIH National Center for Research Resources, Grant No. P41 RR 02170, and Department of Energy ASCI program, National Science Foundation grant ACR-9876914.

Figure 14: *These clouds are shaded with multiple light sources to approximate skylight.*

6.1 References

For more information, updates, lecture notes, and demos, see http://www.cs.unc.edu/~harrism/clouds, and http://www.cs.unc.edu/~harrism/SkyWorks. Another good reference site for clouds is http://www.vterrain.org/Atmosphere/Clouds/index.html.

[Blinn 1982a] Blinn, J.F. A generalization of algebraic surface drawing. (*Proceedings of SIGGRAPH 1982a)*, ACM Press, 273-274. 1982a.

[Blinn 1982b] Blinn, J.F. Light reflection functions for simulation of clouds and dusty surfaces. Computer Graphics (*Proceedings of SIGGRAPH 1982b)*, ACM Press, 21-29. 1982b.

[Dobashi, et al. 2000] Dobashi, Y., Kaneda, K., Yamashita, H., Okita, T. and Nishita, T. A Simple, Efficient Method for Realistic Animation of Clouds. Computer Graphics (*Proceedings of SIGGRAPH 2000)*, ACM Press / ACM SIGGRAPH, 19-28. 2000.

[Dobashi, et al. 1999] Dobashi, Y., Nishita, T., Yamashita, H. and Okita, T. Using Metaballs to Modeling and Animate Clouds from Satellite Images. *The Visual Computer*, *15*. 471-482.1999.

[Ebert 1997] Ebert, D.S. Volumetric modeling with implicit functions: a cloud is born. *ACM SIGGRAPH 97 Visual Proceedings*. 147.1997.

[Ebert and Parent 1990] Ebert, D.S. and Parent, R.E. Rendering and animation of gaseous phenomena by combining fast volume and scanline A-buffer techniques. Computer Graphics (*Proceedings of SIGGRAPH 1990)*, ACM Press, 357-366. 1990.

[Elinas and Stürzlinger 2001] Elinas, P. and Stürzlinger, W. Real-time Rendering of 3D Clouds. *Journal of Graphics Tools*.2001.

[Gardner 1985] Gardner, G.Y. Visual Simulation of Clouds. Computer Graphics (*Proceedings of SIGGRAPH 1985)*, 297-303. 1985.
[Harris and Lastra 2001] Harris, M.J. and Lastra, A. Real-Time Cloud Rendering. Computer Graphics Forum (*Proceedings of Eurographics 2001)*, Blackwell Publishers, 76-84. 2001.
[Henyey and Greenstein 1941] Henyey, L.G. and Greenstein, J.L. Diffuse Radiation in the Galaxy. *The Astrophysical Journal*, *90*. 70-83.1941.
[Kajiya and Von Herzen 1984] Kajiya, J.T. and Von Herzen, B.P. Ray tracing volume densities. Computer Graphics (*Proceedings of SIGGRAPH 1984)*, ACM Press, 165-174. 1984.
[Lewis 1989] Lewis, J.P. Algorithms for solid noise synthesis. Computer Graphics (*Proceedings of SIGGRAPH 1989)*, ACM Press, 263-270. 1989.
[Maciel and Shirley 1995] Maciel, P.W.C. and Shirley, P. Visual navigation of large environments using textured clusters. (*Proceedings of Symposium on Interactive 3D Graphics 1995)*, ACM Press, 95 - ff. 1995.
[Max 1995] Max, N. Optical Models for Direct Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, *1* 2. 99-108.1995.
[Mie 1908] Mie, G. Bietage zur Optik truver medien Speziell Kolloidaler Metallosungen. *Annallen der Physik*, *25* 3. 377.1908.
[Miyazaki, et al. 2001] Miyazaki, R., Yoshida, S., Dobashi, Y. and Nishita, T. A Method for Modeling Clouds Based on Atmospheric Fluid Dynamics. (*Proceedings of The Ninth Pacific Conference on Computer Graphics and Applications 2001)*, IEEE Computer Society Press, 363-372. 2001.
[Nagel and Raschke 1992] Nagel, K. and Raschke, E. Self-organizing criticality in cloud formation? *Physica A*, *182*. 519-531.1992.
[Nishita, et al. 1996] Nishita, T., Dobashi, Y. and Nakamae, E. Display of clouds taking into account multiple anisotropic scattering and sky light. Computer Graphics (*Proceedings of SIGGRAPH 1996)*, ACM Press, 379-386. 1996.
[Overby, et al. 2002] Overby, D., Melek, Z. and Keyser, J. Interactive Physically-Based Cloud Simulatin. (*Proceedings of Pacific Graphics 2002)*, 469-470. 2002.
[Perlin 1985] Perlin, K. An Image Synthesizer. Computer Graphics (*Proceedings of SIGGRAPH 1985)*, ACM Press, 287-296. 1985.
[Reeves 1983] Reeves, W. Particle Systems - A Technique for Modeling a Class of Fuzzy Objects. Computer Graphics (*Proceedings of SIGGRAPH 1983)*, ACM Press, 359-375. 1983.
[Reeves and Blau 1985] Reeves, W. and Blau, R. Approximate and probabilistic algorithms for shading and rendering structured particle systems. (*Proceedings of SIGGRAPH 1985)*, ACM Press, 313-322. 1985.
[Schaufler 1995] Schaufler, G. Dynamically Generated Impostors. (*Proceedings of GI Workshop "Modeling - Virtual Worlds - Distributed Graphics" 1995)*, infix Verlag, 129-135. 1995.
[Shade, et al. 1996] Shade, J., Lischinski, D., Salesin, D.H., DeRose, T. and Snyder, J. Hierarchical image caching for accelerated walkthroughs of complex environments. (*Proceedings of SIGGRAPH 1996)*, ACM Press, 75-82. 1996.
[Strutt 1871] Strutt, J.W. (Lord Rayleigh). On the light from the sky, its polarization and colour. *Philos. Mag.*, *41*. 107-120, 274-279.1871.