GPU-Assisted High Quality Particle Rendering

Deukhyun Cha†, Sungjin Son‡, and Insung Ihm†
†Department of Computer Science and Engineering, Sogang University, Korea
‡System Architecture Lab., Samsung Electronics, Korea

Abstract
Visualizing dynamic participating media in particle form by fully solving equations from the light transport theory is a computationally very expensive process. In this paper, we present a computational pipeline for particle volume rendering that is easily accelerated by the current GPU. To fully harness its massively parallel computing power, we transform input particles into a volumetric density field using a GPU-assisted, adaptive density estimation technique that iteratively adapts the smoothing length for local grid cells. Then, the volume data is visualized efficiently based on the volume photon mapping method where our GPU techniques further improve the rendering quality offered by previous implementations while performing rendering computation in acceptable time. It is demonstrated that high quality volume renderings can be easily produced from large particle datasets in time frames of a few seconds to less than a minute.


1. Introduction
1.1. Background and our contribution
Particles are an excellent means of representing dynamic and fuzzy effects of natural phenomena. They are frequently used by animators to effectively model participating media such as smoke and clouds. The generation of physically accurate rendering images from large particle datasets by considering the complicated light transport phenomena often leads to a substantial expense in computation. Among the several direct particle rendering methods that have been employed in the computer graphics community, one effective approach is transforming particle data into corresponding volume data and then applying an advanced volume rendering algorithm that permits effective massively parallel simulation of the light transport theory on today’s many-core processors, generating high quality renderings in practical time.

In this paper, we propose several computing techniques for particle volume rendering that can be easily accelerated by the current graphics processing unit (GPU). Our method is based on the volume photon mapping algorithm [Jen01] that is routinely used to produce high quality volume rendering images. We first present a three-pass density estimation method, exploiting the GPU’s fast rectangle drawing capability, to use adaptive smoothing lengths for transforming given large particle datasets into volumetric density fields that are then visualized in the following rendering pipeline. Second, we demonstrate that the mathematically correct numerical method for tracing photons in nonhomogeneous media can be implemented practically on the GPU, faithfully simulating multiple scattering, which is one of the most intractable phenomena to reproduce in volume rendering. Then, we show that the speed of volume rendering can be increased drastically by caching precomputed in-scattered radiances in a simple 3D grid structure, and reusing them for fast radiance resampling. Finally, we show that noise can be naturally and efficiently included in the ray marching stage of our GPU-assisted volume photon mapping framework, which allows animators to easily model fuzzy appearances of participating media at very small expense.

The fundamental goal of this work is to provide animators in the special effects industry with a fast volume rendering tool that can significantly increase the efficiency of their animation process. Unlike some of the recent works that pursue real-time volume rendering by possibly performing substantial preprocessing and/or employing simpler light transport models, the presented rendering scheme takes the other direction, attempting to further improve rendering quality provided by the frequently used volume photon mapping technique. We demonstrate that high quality volume renderings

submitted to Eurographics Symposium on Rendering (2009)
can be easily created from large particle datasets in time frames of a few seconds to less than a minute.

2. Previous work

Since introduced to the computer graphics community [Rec83], particles have been employed effectively to represent dynamic and fuzzy phenomena of participating media (please refer to the survey paper by Cerezo). In the physically-based fluid simulation, the Lagrangian particles have also been explored as a means of complementing the Eulerian grid-based approaches [Mon88]. In many particle systems, the rendering computation was usually performed by splatting particles one by one onto screen. While often satisfactory, such direct particle drawing methods were not very appropriate for creating global illumination effects.

Following the pioneering works [Bli82, KH84], there have been extensive studies on the realistic rendering of participating media (please refer to the survey paper by Cerezo for a comprehensive overview of this theme [CPP05]). Among them, volume photon mapping has been frequently used because of its simple and practical simulation of the complex light transport process through volume photon maps [JC98, Jen01]. In order to ease the computational overhead of the volume photon mapping method, the idea of radiance caching was recently extended to participating media [JDDZ08]. Photon tracing was also performed on the face-centered cubic lattice to simulate the light transport phenomena [QXF08, BNM08, W07].

In an aim to achieve interactive or real-time rendering of participating media, various approximation models were implemented on modern graphics hardware [DYN02, KPh03, REK04, SRNN05]. Fast rendering was also possible through precomputations of light transport information [HL01, PARNN04, HAP05, SKSU05, ZRL08, BNM08, CNLE09].

3. GPU-assisted particle rendering

3.1. Rendering pipeline overview

The presented particle rendering scheme consists of four major stages: density volume generation, volume photon tracing, illumination cache construction, and finally ray marching. It starts by transforming the input particles into a volumetric density field for efficient rendering computation on the GPU. Particles are an excellent means for modeling participating media. From the perspective of physically-based volume rendering that needs repeated resampling operations over sampled density data, however, grids are better suited for GPU computation than particles, as the GPU has been optimized for grid structures. For dynamic particle datasets, the conversion has to be carried out for each animation frame, demanding a fast transformation process. In order to accelerate it, we used a rectangle drawing-based method that provides both fast and adaptive conversion on the GPU.

Then, a volume photon map was built by tracing photons stochastically within participating media. While most participating media we routinely handle are nonhomogeneous, a simple propagation distance estimation model that only works for homogeneous media has been usually applied for practical volume rendering, often producing erroneous illumination effects. In order to achieve high quality renderings, we solved on the GPU the integral equation derived for nonhomogeneous media, which offers mathematically correct simulation at small extra cost.

When the volume photon map was ready, radiance estimation computation was performed before ray marching started, by building a 3D grid structure, called illumination cache. This grid, which stored sampled radiance in-scattered towards the viewing direction, was then exploited by the GPU to quickly reconstruct incoming radiance at sampled points through trilinear interpolation in the ray marching stage. Finally, ray marching computation was performed to generate the final rendering images, efficiently utilizing the density volume and illumination cache. In this ray marching stage, we demonstrated that clever use of precomputed Perlin noise could add fine detail to particle data at insignificant expense, which enabled animators to easily model the fuzzy, natural looking appearance of participating media.

3.2. Adaptive density volume generation

In smoothed particle hydrodynamics (SPH), the physical properties of a given particle system are reconstructed using the kernel function [Mon88]. In particular, the density of particle \( p \) is estimated at its location as a weighted sum \( \rho(p_a) = \sum q m_q \cdot W(|p_a - p_q|, h_p) \) over neighboring particles \( q \) with mass \( m_q \), where \( p_a \) is the location of particle \( a \), and \( W(x, h) \) is a smoothing kernel with smoothing length of \( h \) and circular support domain with radius \( \kappa h \) (note that the value of \( \kappa \) may not be one).

By evaluating the weighted sum at grid points, particle data can be easily transformed into corresponding volumetric data. An important parameter in this process that has significant influence on the final renderings is the smoothing length \( h_{ijk} \), which is applied at each grid point \( p_{ijk} \). It is known that local accuracy achieved by density estimation depends on the number of neighboring particles involved. When too few are used, it may result in low accuracy. On the other hand, if too many are used, local properties may be smoothed out despite the high computational effort. In order to gain a consistent level of accuracy throughout the entire simulation domain, the smoothing length should be chosen by reflecting the local particle density, so that a sufficient and necessary number of particles are used.

Our rendering system provides both uniform and adaptive smoothing length methods that are implemented using sim-

submitted to Eurographics Symposium on Rendering (2009)
ple rectangle drawing that is easily accelerated on the GPU. The single-pass uniform length method that has been frequently employed in various computer graphics applications is computationally cheap, whereas the multi-pass adaptive method offers automatic adaptation for datasets of irregular distribution with steep density contrasts.

Note that the kd-tree structure that provides effective nearest neighbor searching may be employed for the adaptive density volume generation. Recently, a kd-tree construction algorithm was presented for the GPU using the CUDA API [ZHWG08]. In this work, an iterative $k$-nearest neighbor search algorithm was implemented based on range searching because the often-used priority queue method [Jen01] was not well-suited for the CUDA’s computing architecture. While proven to be effective for caustics rendering, their construction algorithm is somewhat complicated to program on the GPU. In stead of implementing the kd-tree method, we decided to develop an approximate searching methods that, whether adaptive or not, demands drawing of only $O(N)$ rectangles for $N$ particles (under the assumption that $h(0)$ is small compared with the entire simulation domain) and allows a much simpler implementation on the GPU than the kd-tree approach.

### 3.2.1. Density estimation with uniform smoothing length

Suppose that a volume with resolution of $n_x \times n_y \times n_z$ is to be constructed with uniform smoothing length $h$. In the preparation step, input particles traverse the CPU to create $n_z$ bins, one per $z = z_i$ slice ($0 \leq i < n_z$), each holding particles that may affect grid points on the corresponding $xy$ slice. Stored as vertex arrays, the particle bins are sent to the GPU for fast processing, where particles are orthogonally projected bin-by-bin onto the image plane by drawing squares of size $2kh$, centered at their respective positions in the volume space. For each rasterized pixel that corresponds to a grid point, the pixel shader calculates the distance to the current particle, and accumulates the appropriate kernel-weighted mass. After the rectangle drawing process is repeated over all bins, the resulting $n_z$ images are assembled into a volumetric density field in the form of a 3D texture.

### 3.2.2. Density estimation with adaptive smoothing length

For adaptive selection of smoothing length, we have devised a predictor-corrector scheme that proceeds in three steps. In this method, the smoothing length is controlled by a user-specified parameter $N^*$, which intends that a rough number of particles be involved at each grid point $p_{ijk}$ in the density estimation. In order to avoid a smoothing length that is too large in low-density region, it is limited by the maximum length $h(0)$, which is also set by the user.

1. **[Step 1: Prediction]** In the beginning, one rectangle drawing pass is carried out with an initial length $h(0)$, counting the number of particles, $N_{ijk}^{(0)}$ that exist around grid point $p_{ijk}$ in the spherical volume with radius $kh(0)$. After the particle projection is finished, the average density at $p_{ijk}$ is estimated as
   \[ \bar{\rho}_{ijk} = \frac{mN_{ijk}^{(0)}}{4\pi k^3}, \]
   under the assumption that the mass $m$ is constant. By letting $\tilde{\bar{\rho}}_{ijk} = \frac{mN_{ijk}^{(0)}}{4\pi (kh(0))^3}$, a new smoothing length $h_{ijk}^{(1)}$ that, hopefully, contains $N^*$ particles in the supporting domain is estimated as
   \[ h_{ijk}^{(1)} \equiv \min \left\{ h_{ijk}^{(0)} \left[ \frac{\tilde{\bar{\rho}}_{ijk}}{N_{ijk}^{(0)}} \right], h_{ijk}^{(0)} \right\}. \]

2. **[Step 2: Correction]** The estimate $h_{ijk}^{(1)}$ is reliable only when particle distribution is varying smoothly. In reality, particles are distributed in a complicated and irregular manner, hence a correction computation must be performed to improve accuracy. In this second step, another round of rectangle drawing is performed, counting the number of particles, $N_{ijk}^{(1)}$ found with $h_{ijk}^{(1)}$. When the smoothing length has been underestimated, that is, $N_{ijk}^{(1)} < N^*$, $h_{ijk}^{(1)}$ must be increased appropriately. The correction can be done from a simple relation
   \[ \frac{N_{ijk}^{(0)} - N_{ijk}^{(1)}}{N_{ijk}^{(1)}} \approx \frac{\bar{\rho}_{ijk}^{(0)} - \bar{\rho}_{ijk}^{(1)}}{\bar{\rho}_{ijk}^{(1)}}, \]
   leading to a new estimate
   \[ h_{ijk}^{(1)} \equiv \left( \frac{(N^* - N_{ijk}^{(1)}) ((h(0))^3 - (h_{ijk}^{(1)})^3)}{N_{ijk}^{(0)} - N_{ijk}^{(1)}} + (h_{ijk}^{(1)})^3 \right)^{\frac{1}{3}}. \]

3. **[Step 3: Density computation]** In the final step, the volume density field is generated by performing one more pass of rectangle drawing using the adaptively chosen smoothing distance $h_{ijk}^{(1)}$.

### 3.3. Test results

Table 1 shows timing statistics that compare the run times of several methods. First, we coded a simple uniform grid method (UG), similar to the GPU implementation designed for photon mapping [PDC03], where particles are organized into grid cells of size $kh$ and the cell containing given volume grid point $p_{ijk}$ and its 26 neighbors are queried. Then, this uniform smoothing length method was tested over our rectangle drawing techniques, where USL and ASL3
respectively denote the uniform and adaptive methods described in this section. We also programmed a two-pass adaptive method (ASL2) that skips the correction step, and hence is faster, but possibly less accurate.

As the test results indicate, our uniform length method (USL) compares very favorably with the uniform grid method (UG) that usually took longer than the sophisticated, three-pass adaptive method (ASL3). The run time of our multi-pass method greatly depends on the initial choice of \( h^{(0)} \) that determines the rectangle size. Recall that our scheme controls the overall accuracy of density estimation using the target particle number, \( N^* \). Initially, \( h^{(0)} \) is set large enough, so that each grid point holds at least \( N^* \) particles in its supporting domain. A careless, fixed choice of \( h^{(0)} \), however, may result in a waste of time as it can be too large for most of the grid points. It is possible to select \( h^{(0)} \) heuristically by applying a simple and fast histogram technique on the CPU. The figures in the ASL3 column indicate run times obtained when \( h^{(0)} \) was adaptively set to 0.75\( h^{(0)} \) in the high-density region, in which the statistics in Table 2 are the same as when fixed \( h^{(0)} \) was applied.

In addition to the timing performance, we have also investigated the actual number of particles used by our multi-pass methods. In Table 2, \( N_{\text{avg}}^{\text{used}} \) and \( N_{\text{std}}^{\text{used}} \) respectively indicate the average and standard deviation of the number of particles found at grid points for density estimation. As expected for the uniform smoothing length method (USL), the average numbers increased quickly as \( h^{(0)} \) became larger, demanding more computing time. When the search radius was small, the standard deviation was large, entailing an uncontrolled accuracy in density estimation. The variation was reduced when a larger smoothing distance was used, but too many particles tended to be found in the high-density region, which caused highly detailed features to be blurred. This phenomenon was quite expectable because the tested particle data have a very irregular distribution with steep density contrasts, as displayed in Figure 1.

<table>
<thead>
<tr>
<th>Resolution</th>
<th>( h^{(0)} ) (unit: sec.)</th>
<th>USL</th>
<th>UG</th>
<th>USL</th>
<th>ASL2</th>
<th>ASL3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1290 \times 1030 \times 1070</td>
<td>0.96</td>
<td>1.38</td>
<td>0.53</td>
<td>0.92</td>
<td>1.20</td>
<td></td>
</tr>
<tr>
<td>2570 \times 2060 \times 213</td>
<td>0.96</td>
<td>5.48</td>
<td>1.84</td>
<td>3.55</td>
<td>4.66</td>
<td></td>
</tr>
<tr>
<td>2570 \times 2060 \times 213</td>
<td>1.44</td>
<td>10.02</td>
<td>4.53</td>
<td>7.47</td>
<td>9.98</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Timing statistics measured on an NVIDIA GeForce GTX 280 GPU. A physically-based simulated scene in Figure 1 that contains 326,752 particles was tested to compare run times of various density estimation methods. The uniform grid (UG) and uniform smoothing length (USL) methods used fixed smoothing length. Despite being faster than the other adaptive methods, they were unable to adapt themselves to the density of the particle data. With additional expenses, the presented three-pass method (ASL3) was capable of adaptively controlling smoothing lengths, entailing enhanced renderings, as shown in Figure 1. All the timings contain the CPU time spent for building the uniform grid and slice bins, respectively.

By applying the three-pass adaptive method (ASL3), however, we gained much more control over the number of particles, as shown in the \( N_{\text{avg}}^{\text{used}} \) and \( N_{\text{std}}^{\text{used}} \) columns. In particular, the figures in the ASL2 and ASL3 rows indicate that the additional correction step deserved extra computing time. It may seem that the standard deviation is still high. Considering that the tested particle dataset was very irregular in its distribution, however, we believe that a good enough control for effective rendering was achieved. This fact is well observed in the rendering images in Figure 1. When a fixed length method is used, animators usually have to go through a trial and error process of finding the proper fixed smoothing length. On the other hand, our multi-pass methods automatically control the smoothing length, given an initial value of \( N^* \), by appropriately reflecting the particle density (see the caption of Figure 1).

3.4. Volume photon tracing for nonhomogeneous media

3.4.1. Solving integral equation for distance generation

When a volume photon is scattered at location \( p_0 \in \mathbb{R}^3 \) in a given participating medium, a pair of random direction \( \theta_{\text{next}} \) and distance \( d_{\text{next}} \) to the next interaction point must be stochastically generated. \( \theta_{\text{next}} \) is usually computed by importance-sampling phase function set to the medium. On the other hand, \( d_{\text{next}} \) has often been generated in volume rendering using uniform random variable \( \xi \in (0, 1) \) as

\[
d_{\text{next}} = -\log \frac{\sigma_t}{\sigma_e} \quad (\text{Eq. (1)}),
\]

where \( \sigma_t \) is the extinction coefficient at \( p_0 \) [Jen01].

This expression assumes an exponential distribution with the fixed rate parameter \( \sigma_t \) along the half-closed line \( p[x) = 0 \)
The integral equation as follows:

\[ p_0 + x \cdot \hat{\sigma}_{ext} (x \geq 0), \]

implying that \( \sigma_t \), the extinction coefficient in volume rendering, must be constant. In general, this assumption does not hold because the extinction coefficient varies very irregularly in nonhomogeneous media, leading to erroneous simulation of light transport phenomena. Based on the principle of the inversion method, the propagation distance \( d_{next} \) for nonhomogeneous media with varying extinction coefficient \( \sigma_t(p(x)) \) can be generated by solving an integral equation as follows: For a uniform random number \( \xi \) between 0 and 1, find \( d_{next} \) such that

\[ \int_0^{d_{next}} \sigma_t(s) \, ds = -\ln(1 - \xi) \quad (Eq. (2)). \]

This integral equation can be solved numerically by incrementally evaluating the integral along the line \( p(x) \) until the sum exceeds \(-\ln(1 - \xi)\) (this corresponds to marching along the parameterized line). If the accumulated value has not reached \(-\ln(1 - \xi)\) when the parameter variable \( x \) arrives at the boundary of volume, it means that the traveling photon leaves the medium without an interaction.

Photon propagation by Eq. (2) has already been reported in previous works, for instance, [JC98, PKK00]. To the best of our knowledge, however, the simple exponential distribution is still in popular use for practical volume rendering, probably due to its computational simplicity (for example, path distance was sampled according to the exponential distribution in a recent Monte Carlo simulation of multiple scattering in nonhomogeneous participating media [JDZJ08]). However, the massively parallel computing power of the recent GPU enabled us to employ the sophisticated numerical model that permits more precise light simulation in volume rendering, as will be analyzed shortly.

3.4.2. Test results

As demonstrated in Figure 2, the stochastically correct scheme compares very favorably with the previous simple one. In this example, a cloud ball of low density, containing three smaller chunks of higher densities, was lit by a light source from above. When photons were traced with Eq. (1) in the cloud whose extinction coefficient was set to its density (OLD1), most propagation distances were overestimated because of the small value of \( \sigma_t \) in the low-density region, forcing the photons to leave the cloud without interacting with the thicker regions. This phenomenon was well reflected in the experimental statistics, where more than four million photons had to be emitted to store only 117,757 volume photons (see the OLD1 column in the table in Figure 2(g)). On the other hand, the mathematically correct method (NEW) nicely reflects the density variation of the cloud ball, as demonstrated in the images in (c) and (d). Intuitively correct, photons were stored, stochastically well distributed according to the density distribution, catching the illumination effect properly.

We have also varied \( \sigma_t \) by scaling the extinction coefficient by a factor of 50, and applied Eq. (1) in the photon tracing stage (OLD50). In this setting, too many photons were stored in the upper region of the cloud where light enters, since the small distance values due to the exaggerated extinction coefficients cause most photons to be absorbed without reaching the lower region. Even with the large number of photons stored (416,219 in the OLD50 column of the table in (g)), the two thick balls at the bottom were not illuminated properly, whereas only 191,636 photons as given in the NEW column of the table, were able to correctly create the light scattering effect by solving the integral equation (compare the images in (f) and (d)).

It may seem that the numerical method that needs an inversion of the integral function is much slower than the method based on Eq. (1), as an integral equation must be numerically solved every time a photon interacts with the participating medium. However, as indicated by the run times given in the table in (g), our test on the GPU revealed that the mathematically correct technique is very competitive, especially for intractable situations, because fewer photons were sufficient to faithfully simulate the light transport phenomena. For precise evaluation of the integral, we applied the fourth-order composite Simpson’s rule for the integral equation.
Figure 2: Comparison of stochastic distance generation methods.

The complicated technique (NEW), based on Eq. (2), effectively depicted the illumination effect in a tricky situation as set up in this example, whereas the simpler technique using Eq. (1) had trouble in photon tracing. The timings, measured on an NVIDIA GeForce GTX 280 GPU show that the numerically more expensive method actually turned out to be faster, as it was able to effectively simulate the light transport phenomena with a smaller number of photons. In our implementation, photons were traced on the GPU by drawing rectangles repeatedly and the fourth-order composite Simpson’s rule was applied with step size of one eightieth of the diagonal distance of a grid cell.

\[
\sum \sigma_t(x_j) + 4\sigma_t(x_{j+1}) + \sigma_t(x_{j+2}),
\]

which runs as fast as less precise, low-order numerical methods on the GPU.

### 3.5. Single and multiple scattering on the GPU

#### 3.5.1. Illumination cache

In volume photon mapping, most of the rendering time is spent in the final, ray marching stage, in which in-scattered radiance must be repeatedly estimated at each ray sample point. Usually, the single scattering part of incoming radiance is separately evaluated by performing another ray marching toward light sources. On the other hand, the radiance due to multiple scattering is approximated by summing weighted fluxes of neighboring photons.

In an attempt to enhance computational efficiency, a radiance caching algorithm [JDZJ08], was recently presented for volume rendering, where global illumination within participating media was simulated by sparsely sampling the single and multiple scattering radiances and caching them for subsequent extrapolation. While novel and effective, the algorithm is not best suited for GPU implementation because managing the caching structure, for instance, is still too complex for the computational model offered by the current GPU.

For simple and efficient computation on the GPU, we used a 3D grid, called illumination cache, to store view-dependent, sampled incoming radiance. Once built into a texture image, in-scattered radiance at ray sample points can be efficiently reconstructed through trilinear interpolation, easily accelerated by texture units of the GPU. In building the illumination cache, we separately accumulate the two radiances from single and multiple scattering. In particular, when the radiance due to multiple scattering was estimated, we applied the rectangle drawing technique, proposed for the density estimation computation, for adaptive volume radiance estimation, rather than implementing an extra search structure like the kd-tree for finding neighboring photons.

#### 3.5.2. Test results

Table 2 lists statistics regarding illumination cache, gathered on a PC with a 3.16 GHz Intel Core 2 Duo CPU and an NVIDIA GeForce GTX 280 GPU. To demonstrate speedups achieved by the presented GPU technique, we have implemented a CPU version of the same rendering algorithm that also employs the illumination cache, built using the kd-tree method. During ray marching for both single scattering accumulation and final ray marching, we used a sampling distance that is one third of the diagonal distance of a grid cell.

As the table in (a) indicates, the single scattering computation, based on the fourth-order composite Simpson’s rule, is remarkably efficient on the GPU, achieving computation times that were up to hundreds times faster than on the CPU, because the simple ray marching operation from each grid point of illumination cache was well suited to the GPU’s massively parallel streaming computation architecture. The GPU computation also showed substantial speedups over the CPU when multiple scattered radiance was accumulated onto the illumination cache, as shown in table (b). Here, the performance gap between two processors became larger for a bigger search radius, because the rectangle drawing method on the GPU outperformed the software-based kd-tree operation. Finally, the table in (c) shows how effectively the presented illumination cache worked on the GPU, where the figures in parentheses denote the total rendering time. We have also coded another CPU version that did not utilize
The CPU and GPU codes implementing basically the same illumination cache-based rendering algorithm were compared. In (b), the figures in parentheses in the Resolution column denote the maximum search radius used during volume photon gathering, while the other parenthesized numbers represent the respective numbers of photons stored in the volume photon map. In (c), the figures in parentheses in the Resolution column indicate numbers of jittered supersamples per pixel, and the other parenthesized numbers represent total rendering times including the density volume generation. All timings were measured in seconds.

### 3.6. Ray marching with Perlin noise

Procedural noise has been a very useful tool in volume rendering for increasing the appearance of realism. For instance, volumetric density fields for very natural-looking clouds were generated by adding details through noises to roughly shaped cloud models represented by a set of implicit functions [EMPP03]. The same idea could be easily applied to our GPU-assisted volume rendering scheme in the density estimation stage of our GPU-assisted particle rendering scheme, where pregenerated noise was stored as a texture image for fast access. Another computation point where noise can effectively add extra small-scale detail to already perturbed density fields is in the final ray marching stage.

It has been noted that aliasing artifacts can be reduced by jittering sample points [Jen01]. In fact, we observe that slightly dispersing sample points with noise is a very good way of producing fuzzy effects, particularly for simple modeling data that animators routinely generate, because such particle datasets usually do not contain much detail of participating media, unlike those that are generated by physically-based simulations (see Figure 3). For efficient jittering of sample points on the GPU, we generated offset noises at grid points using Perlin noise in the preprocessing stage, and stored the offset vectors with density value in a four-component 3D texture. Since offset noise to a given sample point is quickly fetched from the texture through trilinear interpolation, the cost of exploiting this position noise in the ray marching step is very cheap.

![Ray marching with Perlin noise](image)

**Figure 3:** Perlin noise in ray marching (Cloud2). By slightly jittering sample points during ray marching ((d)), finer detail can be easily added to density in a field that has already been perturbed with noise ((c)). Since the respective scalar and vector noises applied to the density estimation and ray marching stages are stored in 3D textures, the extra computational cost of exploiting noise on our GPU-assisted particle renderer is very cheap.

### 4. Experimental results and concluding remarks

We have implemented our GPU-assisted particle rendering method on a PC with a 3.16 GHz Intel Core 2 Duo.
CPU and an NVIDIA GeForce GTX 280 GPU, in which
the OpenGL and Cg APIs were used for shader program-
ing. In order to show its effectiveness, we have also im-
plemented an equivalent, single threaded CPU version that
programmed the same rendering pipeline. Table 4 summa-
izes the statistics collected for three test scenes (Cloud1:
Figure 4, Cloud2: Figure 3, Smoke: Figure 5). The per-
formance results imply that the presented GPU rendering
scheme is promising in that it produced high quality ren-
derings in quite affordable time while handling all of such
advanced rendering features as adaptively varying smooth-
ing length, stochastically correct photon tracing for nonho-
mogeneous media, and noise control. The performance gaps
between the CPU and GPU implementations became larger
as the volume and image resolutions increased.

In both implementa-
tions, employing the il-
 lumination cache is crit-
ical because the running
times were prohibitive
without it. When the
same grid resolution was
used for storing the sam-
ples density and radi-
ance, the caching tech-
nique produced basically no difference in renderings. It
should be mentioned, however, that there was a slight dif-
ference between the images produced by the two processors
mainly due to the different ways of generating random num-
bers needed for the stochastic photon tracing and noise ap-
lication (see Figure 5). However, it was hard to tell if one is
better than the other except that one was much faster than
the other as shown in Table 4.

The table in Figure 6 also gives a general idea on how
the timing performance of our method is affected by the em-
ployed grid resolution. As observed in the table, the volume
photon tracing (VPT) and ray marching (RM) stages were
rather insensitive to the grid resolution since the step sizes
used in the respective numerical integration were the dom-
inant factors in both computations. This is in contrast with
the other two stages (DVG and ICC) in which the density
and radiance had to be computed respectively for each grid
point. While becoming faster, our renderer entailed unde-
sired smoothing of details at lower grid resolutions. Further-

more, when it was too low with respect to the scene domain,
jaggies became visible due to insufficient number of grid
cells. Our test shows that noise in the ray marching stage
is quite useful for reducing such aliasing problem. For the
Cloud2 scene, such jaggies were invisible when the resolu-
tion was 200 × 107 × 201 or higher (note that this resolution
is not that high considering the entire simulation domain).

The presented GPU-assisted particle rendering pipeline

<table>
<thead>
<tr>
<th>Grid size</th>
<th>Cloud1</th>
<th>Cloud2</th>
<th>Smoke</th>
</tr>
</thead>
<tbody>
<tr>
<td>DVG</td>
<td>VPT</td>
<td>ICC</td>
<td>RM</td>
</tr>
<tr>
<td>(a) (0) = 1 (7)</td>
<td>GPU</td>
<td>1.57</td>
<td>1.40</td>
</tr>
<tr>
<td>(a) (0) = 1 (6)</td>
<td>GPU</td>
<td>1.50</td>
<td>0.79</td>
</tr>
<tr>
<td>(a) (0) = 1 (4)</td>
<td>GPU</td>
<td>1.39</td>
<td>1.39</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Image size</th>
<th>Cloud1</th>
<th>Cloud2</th>
<th>Smoke</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) (0) = 1 (7)</td>
<td>GPU</td>
<td>1.57</td>
<td>1.40</td>
</tr>
<tr>
<td>(a) (0) = 1 (6)</td>
<td>GPU</td>
<td>1.50</td>
<td>0.79</td>
</tr>
<tr>
<td>(a) (0) = 1 (4)</td>
<td>GPU</td>
<td>1.39</td>
<td>1.39</td>
</tr>
</tbody>
</table>

Figure 4: Rendering of Cloud1.

Figure 5: Rendering comparison (Smoke). Two rendering results
by the GPU (left) and the CPU (right) are compared.

Figure 6: Timing performance at multiple volume resolutions (in
seconds). The run time for producing a 1024 × 576 image from the
Cloud2 data was measured for different grid resolutions, holding
all other rendering parameters fixed. A part of rendering results are
compared for the three resolutions (a), (b), and (c), where the details
became blurred as the grid resolution got lowered.

Table 4: Overall timing performance. The step-by-step dissection
of computation times taken by the presented rendering pipeline is
shown. In this experiment, the same volume resolution (Grid size)
was applied to the creation of density volume (DVG) and illumina-
tion cache (ICC). The integral equation in Eq. (2) was numerically
solved for photon tracing (VPT). Two image resolutions (Image size)
were tested where the ray marching time was almost proportional
to the number of image pixels as indicated in the RM column. Ex-
cept Cloud2, the adaptive, three-pass method was used for the den-
sity volume generation (DVG) on the GPU. For an adaptive density
generation and photon gathering on the CPU, we implemented the
kd-tree structure for nearest neighbor searching.
permits an easy extension to various interesting rendering effects. Currently, we are including the light emission phenomena of hot gaseous fluids so that such important rendering effects as fire, flame, and explosion can be realistically and efficiently synthesized.

Acknowledgements: This work was supported by the Korea Research Foundation Grant funded by the Korean Government (KRF-2008-313-D00920).

References


Appendix A. Major rendering parameters for our particle volume renderer

### Density Volume Generation

- \( n_x \times n_y \times n_z \): 3D volume resolution (in the paper)
- \( h(0) \): initial value of smoothing length (in the paper)
- \( N \): target particle number (in the paper)
- \( \varepsilon \): scale factor in relaxation coefficient \( \alpha \) (in the paper)
- \( m_p \): mass of particle \( p \) (in the paper)
- \( s_{\text{density}} \): density noise scale factor (\( s_{\text{density}} = 0 \) if no noise is applied)

### Volume Photon Tracing

- \( d_{\text{MAX}} \): maximum distance to the next interaction (optional)
- \( \Delta x \): step size used in numerical integration (in the paper)
- \( \Lambda \): albedo of participating medium
- \( g \): Henyey-Greenstein phase function parameter
- \( \sigma_{\text{ext}} \): scale factor for extinction coefficient \( \sigma_{\text{ext}}(p(x)) = s_{\text{ext}} \sigma_{\text{ext}}(p(x)) \), in the paper
- \( N_{\text{emission}} \): maximum number of photons emitted
- \( N_{\text{maxstored}} \): maximum number of photons stored
- \( N_{\text{maxint}} \): maximum number of photon interactions
Illumination Cache Construction

- \( n_x \times n_y \times n_z \): 3D volume resolution (in the paper)
- \( h \): photon gathering radius factor (in the paper)
- \( s_{photon} \): scale factor for photon power
- \( \Lambda \): albedo of participating medium
- \( s_{extcoeff} \): scale factor for extinction coefficient \( (\sigma_t(p(x)) = s_{extcoeff} \cdot d(p(x)), \text{in the paper}) \)
- \( g \): Henyey-Greenstein phase function parameter
- \( (\Delta x)'' \): step size used for radiance sampling during single scattering computation

Ray Marching

- \( n_w \times n_h \): image resolution
- \( (\Delta x)' \): step size used for radiance sampling during final ray marching
- \( \Lambda \): albedo of participating medium
- \( s_{extcoeff} \): scale factor for extinction coefficient \( (\sigma_t(p(x)) = s_{extcoeff} \cdot d(p(x)), \text{in the paper}) \)
- \( s_{offno} \): offset noise scale factor \( (s_{offno} = 0 \text{ if no noise is applied}) \)