Eighth Hungarian Conference on Computer Graphics and Geometry, Budapest, 2016

GPU based particle rendering system for isotropic participating media

Bálint Remes, Tamás Umenhoffer and László Szécsi

Budapest University of Technology and Economics, Budapest, Hungary

Abstract

Realistic rendering of participating media is a complex and time-consuming task, thus visualization softwares usually make considerable simplifications in the process. We present a physically-based algorithm for simulating absorption, forward-scattering and isotropic multiple scattering of light in such media. The radiometric simulation utilizes the capabilities of graphics hardware in order to achieve interactive frame rates.

1. Introduction

Participating media such as smoke, clouds, fog, fire, explosion and snow plays an important role in games and movies. Although approximating methods exists for rendering these phenomena, the problem itself is a challenging task due to its complexity.

The light transport in participating media can be described as [10]

$$L_{out} = (1-\alpha)\cdot L_{in} + \alpha\cdot\omega\cdot\int_{4\pi} P(\omega,\omega')\cdot L(\omega')d\omega' + \varepsilon$$

where the material is defined with the following properties: ω single scattering albedo, K extinction coefficient, η density, P phase function and K_e emission coefficient. Instead of the first two attributes, sometimes the K_s scattering coefficient and the K_a absorption coefficient is used. The equation's second term is a recursive integral which is used for computing the in-scattering inside the media. Here the phase function determines the material's scattering property based on the angle between the incident and outgoing direction. With the help of these properties we have all the required input for the equation above, since the opacity is defined as $\alpha = 1-e^{-\tau}$, the optical depth as $\tau = ds\cdot K$ and $\omega = \frac{K_s}{K}$ where $K = K_s + K_a$.

The optically thin model can be used for media which are either very thin or transparent. Examples for such media are hot steam or clean air. For these materials simulating single scattering provides sufficient results. In contrast to these, the correct simulation of optically thick media must take the photon collision with more than one of the particles into account. These kinds of media have an albedo close to 1, such as snow or clouds.

In this paper we propose a particle rendering system for rendering optically thick participating media by utilizing the graphics hardware. The advance capabilities of latest graphics harware made it possible to simulate traditionally offline computed lighting effects in real time[9]. Section 2 presents previous works on the topic of this paper. In Section 3 we give a description of our method for rendering isotropic media. Section 4 addresses the performance problems with particle based solutions in general and also specific to our case. Section 5 discusses the implementation performance and results. Finally, Section 6 concludes and describes possible extensions for the future.

2. Previous Work

Numerous visualization methods for participating media have been proposed, yet alone methods for rendering clouds specifically. Due to its simplicity, particle systems are a popular choice for representing and rendering clouds. One of the earliest widely used real-time method is described by Wang[3]. The algorithm starts with an offline preprocessing step, where the artists create the cloud shapes and their particles' attributes and billboards. During the rendering part, the particle billboards are shaded by both a predefined gradient as well as a phenomenon based directional color. Finally, it uses 8 impostors around the camera to ensure the application runs in interactive framerates. This solution was further enhanced by Wenzel[4] by implementing soft clipping and

adding backlighting to achieve glowing edges when clouds are partially covering the sun.

In his dissertation Harris[2] presents a more physically based approach. This method uses a preprocessing step for calculating the illumination of clouds after which the rendering with illuminated data is done in real-time. The illumination step here uses anisotropic multiple forward scattering, which instead of sampling only in the single dedicated path uses samples in a cone along the dominant scattering direction. More recently Hillaire[5] proposed a method based on Wronski's work[6] which emphasizes the importance of physically based rendering. Their method heavily utilizes the graphics hardware to provide both participating media illumination and rendering at interactive framerates. The approximation here also enables the fine-tuning the anisotropy parameter of the Heyney-Greenstein phase function, however the model limits scattering to the first order. The illuminated data are voxelized to clip space aligned 3D textures.

3. Our rendering algorithm

Our method is capable of simulating the illumination of participating media with directional lights. It can handle multiple dynamic light sources in a single scene, thus their position and their properties can be altered during run-time. The media rendered with the method is self shadowed and the isotropic scattering in the media is also accounted for. This method has 4 main steps and during these it uses voxelization of the media as a volume rendering technique.

Algorithm 1 Illumination and rendering

```
1: procedure ILLUM_AND_RENDER
2:     for each lightSource l[i] do
3:         s[i] ← shade(l[i])
4:     g[0] ← gather(s)
5:     for j in 0..mScatter do
6:         g[j%2 + 1] ← multi_scatter(g[j%2])
7:     sort particles w.r.t. cam pos
8:     render(g, particles)
```

Algorithm 1 gives an overview of the whole rendering process. During lines 2-3 a shading step is performed based on each illuminating light source. Each illuminated data then collected to a single gather buffer. On lines 5-6, the multiple scattering is accounted in a filtering pass. Here during each iteration, the previous result is reused. Finally, in order to render the media as a particle system, the particles are first sorted by their distance from the camera, then they splatted on screen using the previously calculated buffer as seen on lines 8 and 9. We further discuss each substep in the following subsections.

3.1. Shading pass

The first step simulates the direct contribution to the media by a single directional light. Here we produce voxelized data containing the attenuation of light due the travel through the participating media. Due to its simplicity, we account for the forward scattering in this pass too.

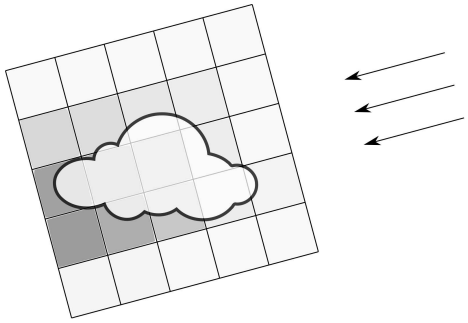

Figure 1: *Voxels of a shaded cloud.*

An oriented bounding volume is used for sampling during voxelization as can be seen in Figure 1. By choosing the orientation of the volume to match the corresponding light source's direction, the shading process can be performed incrementally. In practical applications, voxels are usually stored in 3D textures. Choosing the texture's z (depth) axis to be parallel with the light direction means that the shading process will go through each layer of such texture. Creation of such volume is trivial from the particles' position and dimension data. The goal during this step is to determine the radiance passing through the media, i.e.:

$$\frac{dL}{ds} = -(1 - e^{-K}) + \frac{K_s}{4\pi} P(0^\circ)$$

In practice this can be accomplished using rendering to the layers of the aforementioned texture. Although the particles should be rendered in ascending order based on their distance from the light source, however the equation used for blending is multiplicative, thus the order of draw is not relevant and the sorting of particles can be omitted here.

3.2. Gather pass

In case of multiple light sources, performing each shading pass alone is inadequate. Although it would be sufficient to perform the gathering of shaded volumes at the last moment before the final rendering, running the simulation of multiple scattering on a single volume has reduced the overall computation needed. In practical applications, this also has the benefit of working on a single 3D texture thus requires fewer look-ups during the upcoming pass. This step's goal is to collect the shaded data based on distinct lights to a single volume, as it can be seen on Figure 2.

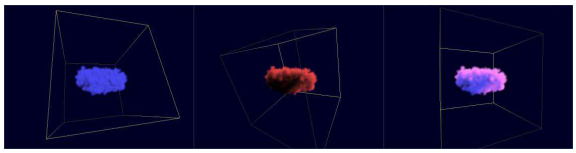

Figure 2: *Shaded media with blue (left) and red (middle) light sources and the gathered result (right).*

3.3. Multiple scattering

The illumination steps so far only considered the attenuation of radiance and single forward scattering. The last step concerning the propagation of light in media aims to simulate multiple scattering. The scattering equation introduced in Section 1 is a recursive integral, which complexity - especially during real-time rendering - is too high for exact computation. Our method assumes the following relaxations: the scattering is only simulated up to a predefined number of steps, only isotropic scattering is considered, and the integral is computed with a finite-element method. In reality, the approximation with a fix number is not that big of an issue as it first seems: the scattered result is dominated by the first and second order scattering only[2]. The FEM samples each voxel's 26 direct neighbors in the following way:

$$L_s \approx K_s \cdot \sum_{i=0}^{26} L(i) \cdot P(\omega_i, \omega)$$

As we previously discussed in section 3, this is an iterative process. In each iteration one step of light scattering is simulated. The input of the function presented in Algorithm 2 is the result voxels produced by the nth step and the output is produced to a different volume with the same amount of voxels containg the $(n+1)th$ step's scattering volume. The input of the zeroth step is the gathered data described in the previous subsection. The probability of scattering is based on the material's albedo and opacity which is then multiplied with the sampled color during the evaluation of sum (lines 4-6). Since only the result of the iteration is relevant from the next iteration's viewpoint, we write the scattering data into a distinct texture. However for the final render we need an accumulated result which contains all scattered data in addition to the gathered result. This accumulated buffer is updated after each iteration on line 8.

3.4. Final rendering

The final step in the process of displaying an illuminated media is the actual render to the screen buffer. This step uses the illumination data created previously. Important to note here that the particles should be rendered in descending order based on their distance from camera. This sort only need to be performed after each movement of camera or media,

Algorithm 2 Simulation of multiple scattering

1: **function** MULTI_SCATTER($gTexIn$)
2: **for** each voxel v in $gTexOut$ **do**
3: $v \leftarrow 0$
4: $msCoeff \leftarrow \frac{v.opacity \cdot albedo}{26}$
5: **for** i in 0..25 **do**
6: $v.color \leftarrow v.color + msCoeff \cdot sampleNeighbor(gTexIn, i)$
7: $v.alpha \leftarrow sampleAlpha(gTexIn, v.pos)$
8: $gTexAccum \leftarrow gTexAccum + gTexOut$
9: **return** $gTexOut$

before the first render. Moreover, as it is common with transparent objects, the drawing of transparent media must occur after the rendering of opaque objects is finished. As with particles, the transparent objects themselves must be rendered in descending order too.

Algorithm 3 Final rendering

1: **procedure** FINAL_RENDER($gTex$)
2: **if** *camera* OR *media* moved **then**
3: sort *particles* w.r.t. *cam pos*
4: $depth\,buffer \leftarrow$ read only
5: $blendingSrc \leftarrow SRC\_ALPHA$
6: $blendingDst \leftarrow 1 - SRC\_ALPHA$
7: $camToLight \leftarrow (camMv^{-1} \cdot cloudMv^{-1} \cdot boundMv^{-1} \cdot minToOrigo \cdot scaleToIdentity)$
8: **for** each *particle* p_i **do**
9: $color \leftarrow$ sample$(gTex, camToLight \cdot p_i.pos)$
10: $alpha \leftarrow 1 - e^{-opbb \cdot density \cdot p_i.size}$
11: render$(color, alpha)$

The algorithm of particle system rendering is described in Algorithm 3. The procedure's input texture contains the accumulated illumination data. If we neglect the multiple scattering this is equivalent with the gathered data (see Section 3.2), otherwise it contains the previous subsection's result. In practice, to sample from the texture, we need to prepare a matrix (line 7), which transforms vectors given by camera coordinate system to the light texture's sampling coordinates. The output's color is determined by the sample from the input texture at the particle's position. The alpha is set according to line 10, where *opbb* is the sample from the particle's opacity billboard.

4. Optimization

Although particle system based rendering is popular in real time applications, the results usually suffer from performance problems. The main issue with such system is the fill rate bottleneck: for each final pixel of a single frame, hundreds or even thousands of particles rendered on top of each other using blending methods, thus the number of writes could easily exceed the hardware capabilities.

This problem usually addressed by reducing the size of the rendering buffer and thereby the number of writes[7]: first we render the particles to a smalerl buffer, then in a post processing step enlarge the result onto the normal screen-sized buffer. Efficient use the texture cache also plays an important part here. Using either compressed textures (e.g.: DXT1) or textures with reduced channel count and overall small particle texture sizes can result in good speedup. The performance problem is usually further addressed by implementing an impostor-based system, where the result is rendered to a single texture buffer which is then displayed instead. The impostor is only refreshed when a delta metric (e.g.: the view angle difference) is reached its threshold.

The shading phase in our algorithm plays a crucial part in the application's performance. As it was presented in subsection 3.1, the process is required to be executed for each light source separately. Furthermore in each iteration, the voxelization - which is implemented using rendering to 3D textures - requires the rendering of particles multiple times: although the rendering viewport is smaller, the overdraw is more significant. Thus we also optimized the shading pass to reduce the fill rate requirement of the solution. This algorithm which is used by our implementation is presented in Algorithm 4. The key point here is copying the previous voxels' results instead of re-rendering all the particles up to the current voxel (line 4), thus the only drawing required is the particles inside the current voxels (line 5). This incremental pass renders the particles only once while in comparison the brute-force method renders $n \cdot \frac{n+1}{2}$ slices.

Algorithm 4 Optimized shading pass

```
1: procedure SHADING_OPT
2:     slice[0] ← lightSource.color
3:     for i in 1..numDepth do
4:         slice[i] ← slice[i − 1]
5:         render_particles_inside(i)
```

5. Results

We have implemented the presented algorithm using the OpenGL/GLSL APIs. During testing, we measured the implemented application's performance on the following hardware: Intel Core i5-4690 3.5GHz, 8 GB RAM, Nvidia GeForce GTX 690. Throughout the two set of runs, all of the light sources were dynamic in order to simulate a possible worst-case scenario for the real time rendering. The screen resolution was 640x480 and all 3D textures used were 128x128x128 during testing. Each run measured the average frame rate, and during both we also tested the effect on performance of increasing the steps for the multiple scattering process.

The cloud used for the first tests contained 38000 particles. During the 5 runs with the parameters above, only the number of light sources were incremented. The average FPS with one light source was 49, with 5 lights the same value was 16.5. The results are shown in Figure 3.

During the second batch of runs, we used a cloud object more commonly appears in real time scenarios: this time the cloud contained 3100 particles. Again, between the two runs only the number of light sources were modified. Similar to the first set of tests, the light sources had a negative impact on performance, but here having only one light source and omitting multiple scattering at all dramatically improved the performance compared to the rest. These results can be also viewed in Figure 3.

In both cases, the increment of the step count during multiple scattering caused a proportional drop in the performance. It is worth noting that while the decrease in performance is there, it is only linear in the number of steps used for the scattering. Finally, the optimization of shading phase described in section 4 resulted in more than 50% performance increase with few scattering steps.

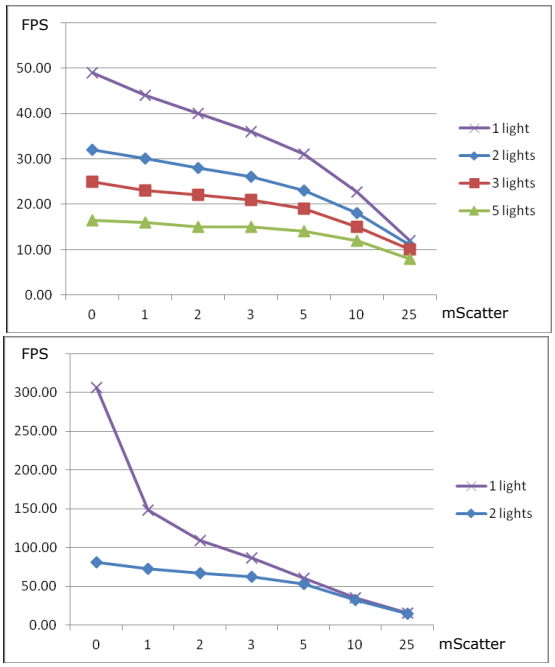

Figure 3: *Results of performance tests with 38000 particles (top) and 3100 particles (bottom).*

6. Conclusions

We have presented a method for illuminating and rendering realistic participating media with multiple dynamic light sources at real-time frame rates. The implemented application supports multiple isotropic scattering as well as gives option to use animated particle-based media. Beside real-time rendering our method can also be used to render media

with massive particle count, which could be used in feature film production (see Figure5).

The topic provides numerous options for possible extensions. A direction aiming to further improve performance could be the transition of particle sorting from CPU to GPU. This would help reducing the performance bottleneck which occurs when rendering objects with huge number (few million) of particles. An other aspect of extension could be the elimination of clipping artifacts during the composition transparent and opaque objects. Such problem can be addressed with the use of depth impostors[8].

Figure 4: *Clouds with multiple environmental light sources (top and middle). Correct self shadowing of concave shapes (bottom).*

Figure 5: *Snow bunny rendered with 1.2 million particles using two light sources and 5 scattering steps.*

Acknowledgements

This work has been supported by OTKA PD-104710.

References

1. L. Szirmay-Kalos, M. Sbert, T. Umenhoffer. *Real-Time Multiple Scattering in Participating Media with Illumination Networks.* Eurographics Symposium on Rendering, 2005.
2. M. J. Harris. *Real-Time Cloud Simulation and Rendering*, 2003. 2, 3
3. N. Wang. *Realistic and Fast Cloud Rendering.*, 2003. 1
4. C. Wenzel. Real-Time Atmospheric Effects in Games. *SIGGRAPH Course 26*, 2006. 1
5. S. Hillaire. *Physically Based and Unified Volumetric Rendering in Frostbite*. *SIGGRAPH*, 2015. 2
6. B. Wronski. *Volumetric Fog: Unified compute shader based solution to atmospheric scattering*. *SIGGRAPH*, 2014. 2
7. H. Nguyen. Fire in the "Vulcan" Demo. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics, Chapter 6*, 2004. 4
8. T. Umenhoffer, L. Szirmay-Kalos. *Real-Time Rendering of Cloudy Natural Phenomena with Hierarchical Depth Impostors.* Eurographics Symposium on Rendering, 2005. 5
9. Szirmay-Kalos, L. and Szécsi, L. and Sbert, M. *GPU-Based Techniques for Global Illumination Effects.* Morgan and Claypool Publishers, San Rafael, USA, 2008 1
10. Szirmay-Kalos, László and Antal, György. and Csonka, Ferenc *Háromdimenziós grafika, animáció és játékfejlesztés* ComputerBooks, Budapest, 2003 1