GPU Ray Marching for Real-Time Rendering of Participating Media

Jeonghun Han
Soongsil University
duckweed@ssu.ac.kr

Hyunwoo Ki
Soongsil University
kih@ssu.ac.kr

Kyoungsu Oh
Soongsil University
oks@ssu.ac.kr

Abstract

This paper presents a GPU based ray marching algorithm for real-time rendering of participating media. We fire a ray at each pixel being shaded on the cube surface, and then we find an intersection between the ray and inner-volume recorded by a 3D texture, using both linear and binary searches. At this intersection, the ray marches through the volume to compute radiance by single scattering of light. This technique can render isotropic as well as anisotropic participating media at tens frames per second on graphics hardware without any preprocessing loads.

Keywords: real-time rendering, participating media, volume rendering, graphics hardware

1. Introduction

It is important to interactively render participating media (e.g., cloud and smoke) as well as other volume data (e.g., CT and MRI) for a variety of applications such as video games, virtual reality, flight simulators, and scientific visualization systems. One way of rendering participating media is ray marching but generally it is CPU intensive. The ray marching can be parallelized at the pixel since rays from each pixel can be traced independently.

Graphics hardware has been rapidly improving its performance, and it gives us high-parallel computation (e.g., NVIDIA GeForce 8800 contains 128 stream processors). We introduce an extension of relief mapping [1], for real-time rendering of participating media using ray marching on commodity graphics hardware.

The relief mapping is a technique for real-time rendering of height-field using per-texel 2D ray casting on the GPU. We extend this ray casting algorithm to a 3D ray marching algorithm. We use a polygonal cube and volume data recorded by a 3D texture, as input data. An eye ray starts from the cube surface in object-space, and the first intersection between the ray and inner-volume is found by both linear and binary searches. At this intersection, we perform ray marching to compute illumination by single scattering employing deterministic sampling of the integration path length. In order to compute the incident path length and irradiance, we also perform both linear and binary searches toward a light. Figure 2 illustrates our algorithm and a simple resultant image.

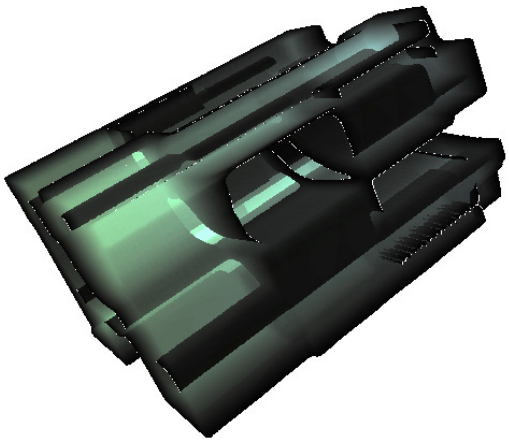

Figure 1. An isotropic engine. It was rendered by single scattering of light and ran at approximately 64 frames per second without preprocessing.

We demonstrate several examples of the proposed technique. We archived real-time performance, approximately hundreds frames per seconds for simple shading and tens frames per seconds for single scattering, on commodity graphics hardware.

2. Related Work

Many single scattering methods for rendering participating media have been presented. Blinn proposed the first visualization technique of

0-7695-2945-3/07 $25.00 © 2007 IEEE
DOI 10.1109/ICCSA.2007.46

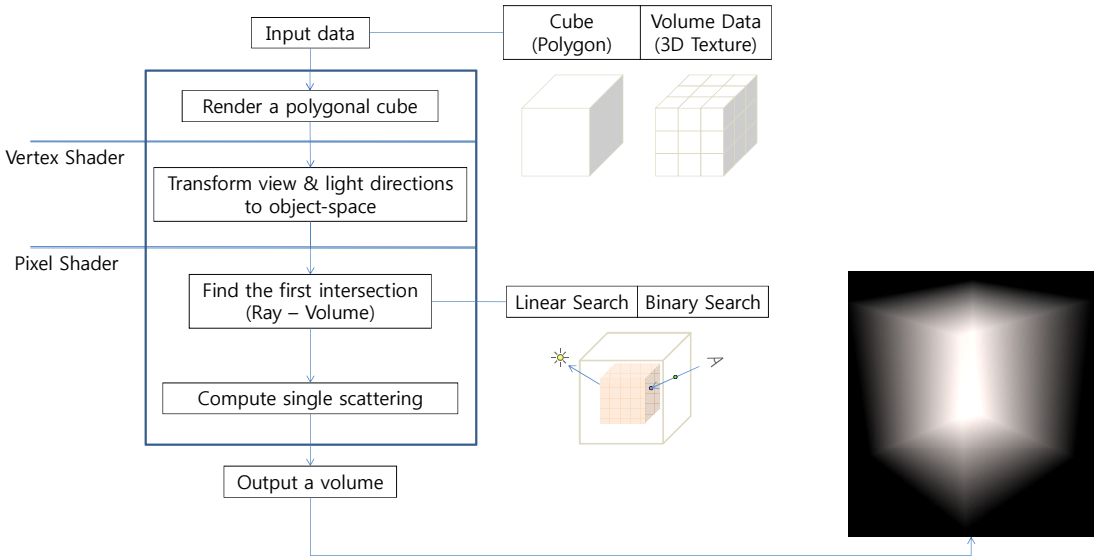

Figure 2. The overview of our algorithm.

participating media [2]. Kajiya and Herzen extended the Blinn's model for ray tracing [3]. Nishita and Nakamae proposed a method for rendering a variety of optical effects within water such as caustic and shafts [4]. These methods, however, consume long computation times. Recently, Sun et al. presented an analytic solution for real-time rendering of participating media such as fog, on the GPU [5].

Another related method is volume rendering, one of the visualization techniques in scientific visualization. The most often used algorithm for high-quality images is ray casting. Ray casting is an image-order traversal of the image plane to find a color and opacity. Levoy et al. accelerate the ray casting using a hierarchical data structure, octree [6]. However, such algorithm is still CPU intensive. Hadwiger et al. proposed a GPU-based isosurface visualization algorithm [7], but their searching algorithm differs from ours and they did not consider realistic single scattering but non-photorealistic rendering.

There are many GPU ray casting algorithms for rendering height-field data. Relief mapping [1] uses both 2D linear and binary searches to enhance accuracy with low cost. These methods are simple and rapid but cannot handle volume data and do not consider single scattering. We extend the relief mapping to 3D ray marching and add fast single scattering.

3. The Transport Equation

This section describes the basic transport equation to understand how we can compute illumination of participating media. The light traveling through a participating medium interacts with its particles, and generally we categorize the interaction into three kinds: emission, absorption, and scattering.

Emission means that how much the radiant energy is created from the medium at the unit distance. Simply, it is represented as a constant value causing incensement of the radiance. While absorption decreases the radiance and changes the radiant energy into other energy forms such as heat, but computer graphics often ignores such changing. These two terms do not consider any angular conditions but only positional ones.

The most important term is scattering, meaning a change in traveling direction. Scattering is generally divided by out-scattering and in-scattering terms. Out-scattering reduces the radiance like absorption, whereas in-scattering increases the radiance like emission. The reduced intensity known as the Beer's law is expressed as $L(x) = L(x_i)\exp(-\sigma_t \|x - x_i\|)$. The exponential term is called the transmittance, $-\sigma_t L dx$ is the optical thickness. Here, $\sigma_t = \sigma_a + \sigma_s$ is the extinction coefficient, and σ_a and σ_s are the absorption and scattering coefficient, respectively.

In-scattering accounts for increased radiance due to scattering from other directions. Especially, the expression for single scattering on homogeneous media

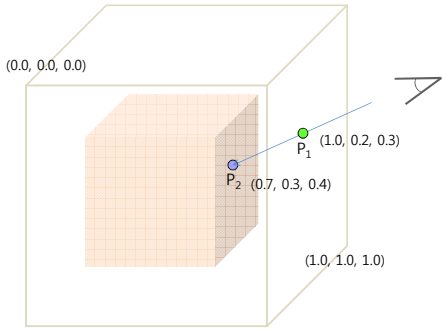

Figure 3. We fire an eye ray from the cube surface and find an intersection between the ray and inner-volume in object-space.

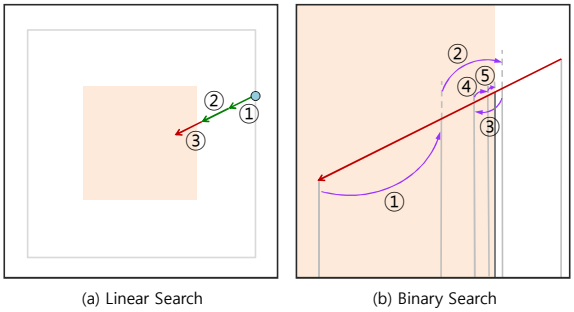

Figure 4. We roughly advance the ray to find a ray-volume intersection linearly (left) and then refine the intersection using a binary search.

is $L_s(x) = \sigma_s \int_0^\infty p(\omega_i, \omega_o) L(x_i) dx$, where p is the phase function that describes the angular distribution of scattered light. If a medium is isotropic, the phase function is the constant of $1/4\pi$.

4. Volume Mapping

Our algorithm consists of finding the volume boundary and computing volume illumination. The input data are a polygonal cube and a 3D texture recorded volume data. In the rendering process, the direction vectors of a view and light are transformed to the cube's object-space in the vertex shader. Searching for the volume boundary and computing illumination are performed in the pixel shader. The first intersection is found by both linear and binary searches of a fired ray from view as shown in Figure 3 and 4. The ray is restarted at that point: it goes linearly and we obtain single-scattered illumination using ray marching. Figure 4 illustrates our algorithm.

4.1. Input Data

A polygonal cube and volume data represented as a 3D texture are the input data. Vertex positions are set from (0.0, 0.0, 0.0) to (1.0, 1.0, 1.0) in object-space. The reason is to easily use the ray position as texture coordinates. Each voxel of the 3D texture has a simple binary value (0 or 1), and it is one if the volume is filled; otherwise, it is zero.

4.2. Linear and Binary Searches in 3D

In order to find the first intersection (entry point) and the last intersection (exit point) between the eye ray and an inner-volume, we perform both the linear search and the binary search. It is different from relief mapping [1]. It is because in the relief mapping the ray goes in 2D tangent-space according to information about 2D height-field to find the first intersection only but our algorithm employs ray marching (i.e., not only the first intersection but also last intersection is needed) in 3D object-space considering inner-volume data.

First, we shoot a ray from the cube surface being shaded. The ray goes linearly as user-defined distance, and for each step a voxel datum is obtained by using the current ray position as the 3D texture coordinates. If the datum is zero, it means that there is not the volume and thus the ray goes. Otherwise, the linear search is stopped because the ray is within the volume. We repeat such linear search until the ray is located in the volume. Some rays will not meet the volume during the searching, and thus we check if the ray exits from the range of zero to one or not, after each iteration. Figure 4.a illustrates this linear search.

Unfortunately, the result by the linear search may be wrong (see Figure 4.a). Therefore, using an additional binary search we find a more accurate intersection after the linear search as shown in Figure 4.b. We assume that there is the exact intersection between the current and previous ray positions (a red arrowed line in Figure 4.b). There are two cases in the binary search: the first case is that the ray is within the volume. In this case, we go back to the outside as a half step. The second case is that the ray is outside the volume. In this case, we advance the ray to the volume as a half step. For a 256^3 texture, we can approximate a good intersection with only 8 steps ($2^8 = 256$). Since we do search within not the entire volume but a small step, we can find an accurate intersection using a few iterations (in our implementation we used only 5 iterations).

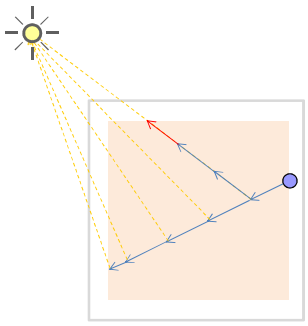

Figure 5. To simulate single scattering, the ray marches to the light linearly. If the ray exits (red line), we perform a binary search to find a more accurate ray-volume intersection.

4.3. Single Scattering

A general approach to simulate single scattering is ray marching. In the ray marching, the ray goes along the line from the first ray-volume intersection to the last one. In practice, to accelerate performance we often stop the ray marching when the transmittance is smaller than a small threshold.

We can easily extend our linear search and binary search to compute single scattering. We assume that the volume is convex. Like the traditional ray marching, we start with the first ray-volume intersection. For each step, we fire a ray to the light direction to compute the incident distance (see Figure 5). We also use both linear and binary searches to find an accurate intersection between the ray exiting and the volume. The search algorithm will be modified slightly: we advance the ray when the ray is within the volume; while we stop searching for the linear search and go back for the binary search when the ray is outside the volume.

After computing the incident distance, we evaluate the phase function using a Schlick's approximation [8] of Heyney-Greenstein's one, $1-g^2/4\pi(1-g\cos j)^2$, where the g is the anisotropy and cos$j$ is the angle between incident and scatted light directions. Then, we solve the single scattering equation with standard Monte Carlo integration. Precisely, we do not move the ray randomly but deterministically (i.e., our linear search step is constant). Such deterministic sense reduces the number of samples significantly, but aliasing will occur. To prevent this aliasing, we use jittering as Pauly et al. [9] did. They used only one random jitter for entire strata. Similarly, we also use a single random jitter but we make the jitter small. That is, [0, 1) to [0.0, 0.2). This reduced both aliasing and noise more efficiently in our experience. Figure 6 shows the effect of jittering. The output color indicates the distance between the first ray-volume intersection and the last one.

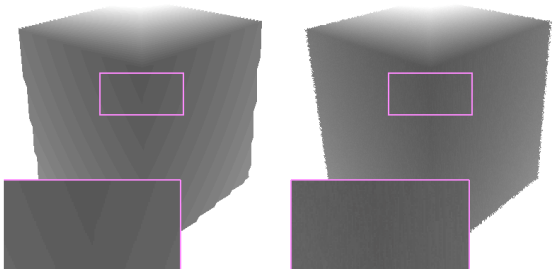

Figure 6. Using the fixed number of searching steps causes aliasing (left). A single small random jitter gives us preventing the aliasing with little noise (right).

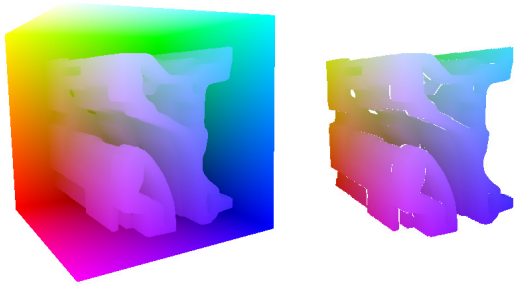

Figure 7. The first ray-volume intersection (left), and peeling a cube by discarding pixels (right).

5. Implementation Issues

We have implemented our algorithm on HLSL 3.0 of Direct3D. We made a simple cube with 8 vertices, and the range of vertex positions are set from (0.0, 0.0, 0.0) to (1.0, 1.0, 1.0). It makes reading a 3D texture easy. In the vertex shader, we transform view and light directions to object-space. Next, in the pixel shader we find the first ray-volume intersection using both linear and binary searches with iterations. After finding the first intersection, we start a linear search from the intersection again. For each step, we perform both linear and binary search to find an intersection of the ray exiting and the volume. We discard pixels when the ray does not meet any volume data (see Figure 7 as an example).

Since current graphics hardware and a shader model cannot generate a random number, we pregenerate random numbers and tabulate these into a 2D texture, and then we read the texture in the pixel shader.

Table 1. Algorithm performance (FPS).

	1st intersection	Distance from 1st to 2nd	Single Scattering
Cube	669	428	92
Engine	1068	679	64

6. Results

This section presents a number of experimental results. All experiments have been done on an NVIDIA GeForce 8800 GTX graphics card and the performance of the PC does not affect results because our method entirely runs on graphics hardware. We use the 512^2 frame buffer and 256^3 volume textures for entire examples.

Figure 1 and an output image in Figure 2 show results rendered by our single scattering. In figure 1 and 7, several holes appear because our current implementation only considers convex objects (i.e., an exiting ray does not go though the volume again). This artifact may be solved using repeating the search algorithm until the ray escapes the not volume but cube.

Table 1 shows the Algorithm performance for a variety of shading types. The results for finding the first intersection are the fastest approximately 669 ~ 1068 fps (frames per second). Computing distance between the first and last intersections spends time about 1.5 times. Single scattering is a very time consuming approach but we were still able to archived interactive rates approximately 64 ~ 92 fps.

7. Conclusion

We have presented a simple method for real-time rendering of participating media by single scattering of light. This can be easily implemented on graphics hardware entirely. In this paper we considered only a homogeneous medium with a single point light source. However, we believe that our method could be extended to support an inhomogeneous medium with multiple lights. Precisely, each voxel stores not a binary value but scalar values for absorption and scattering coefficients, and anisotropy into multiple 3D textures.

Although a binary search enhance quality without costly process, if the step size of a linear search is big or we use very complex and thin volume data then artifacts will occur like relief mapping [1]. To remove the problem we must use more samples for searching but it causes to make the performance slow. We plan to handle a non-convex volume, and a highly complex volume with an adaptive linear search as [10] did for 2D ray casting on height-field data. Recently, Oh et al. proposed a more intelligent and robust 2D ray casting method using a quadtree [11]. We would like to extend their method to a 3D version using an octree like Levoy's spirit [6].

8. Acknowledgments

This work was sponsored and funded by Korea Game Development & Promotion Institute as Korean government project.(Ministry of Culture and Tourism)

9. References

[1] Policarpo, F., Oliveira, M. M., and Comba. J., "Real-time relief mapping on arbitrary polygonal surfaces", *In ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games Proceedings*, ACM Press, 2005, pp. 155-162.

[2] Blinn, J. F., "Light reflection functions for simulation of clouds and dusty surfaces", *In Proceedings of the 9th Annual Conference on Computer Graphics and interactive Techniques*, *SIGGRAPH '82*. ACM Press, New York, NY, 1982, pp. 21-29.

[3] Kajiya, J. T. and Von Herzen, B. P., "Ray tracing volume densities", *In Proceedings of the 11th Annual Conference on Computer Graphics and interactive Techniques*, *SIGGRAPH '84*. ACM Press, New York, NY, 1984, pp. 165-174.

[4] Nishita, T. and Nakamae, E., "Method of displaying optical effects within water using accumulation buffer", *In Proceedings of the 21st Annual Conference on Computer Graphics and interactive Techniques SIGGRAPH '94*. ACM Press, New York, NY, 1994, pp. 373-379.

[5] Sun, B., Ramamoorthi, R., Narasimhan, S. G., and Nayar, S. K., "A practical analytic single scattering model for real time rendering", *In Proceedings of the SIGGRAPH '05,* ACM Press, New York, NY, 2005, pp. 1040-1049.

[6] Levoy, M., "Efficient ray tracing of volume data", *ACM Transactions on Graphics*, Volume 9, Number 3, 1990, pp. 245-261.

[7] Hadwiger, M., Sigg, C., Scharsach, H., Bühler, K., Gross, M., "Real-time ray-casting and advanced shading of discrete isosurfaces", *Computer Graphics Forum,* Volume 24, Issue 3, 2005, pp. 303-312.

[8] Schlick. C., "An inexpensive BRDF model for physically-based rendering", *Proc. Eurographics'94, Computer Graphics Forum*, 13, 3, 1994, pp. 233-246.

[9] Pauly, M., Kollig, T., and Keller, A., "Metropolis light transport for participating media", *In Proceedings of the Eurographics Workshop on Rendering Techniques 2000,* B. Peroche and H. E. Rushmeier, Eds. Springer-Verlag, London, 2000, pp. 11-22.

[10] Tatarchuk, N., "Dynamic parallax occlusion mapping with approximate soft shadows", *In Proceedings of the 2006 symposium on Interactive 3D graphics and games*, ACM Press, 2006, pp. 63-69.

[11] Oh, K., Ki, H., and Lee, C., "Pyramidal displacement mapping: a GPU-based Artifacts-Free Ray Tracing through an Image Pyramid", *In Proceedings of the ACM Symposium on Virtual Reality Software and Technology (VRST '06),* ACM Press, New York, NY, 2006, pp. 75–82.