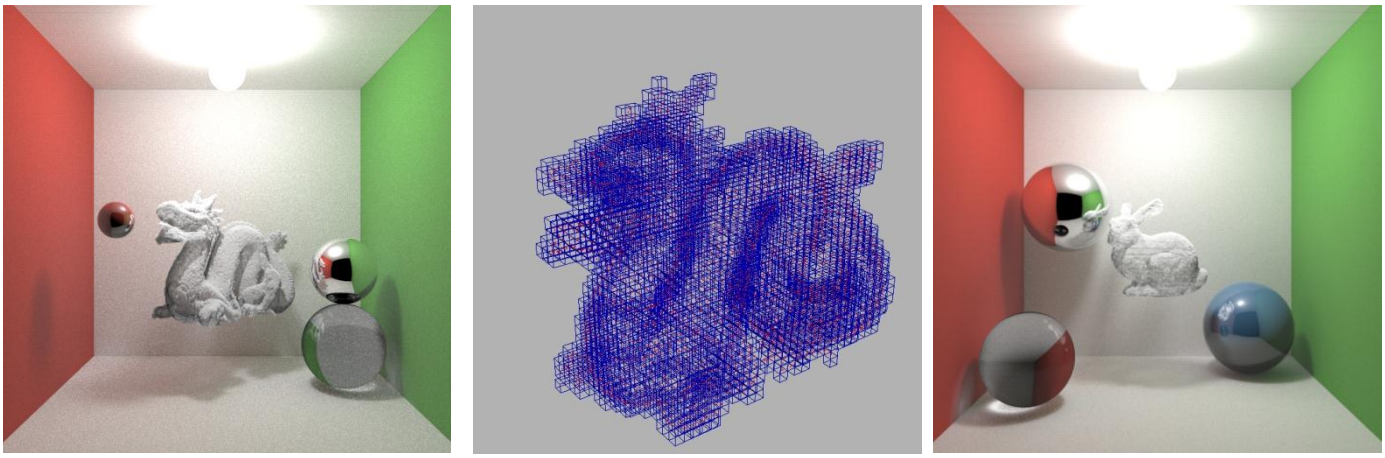
Efficient Volume Rendering in CUDA Path Tracer

Hsuan-Yueh Peng
hsuanyup@usc.edu

Kristopher Wong
kristoph@usc.edu

Kaleb Williams
kalebwil@usc.edu

University of Southern California



a b c

Figure 0: Our volume rendering result and illustration. (a) Stanford dragon rendered as cloud with resolution 0.03.(b) Same model as (a) with our Octree visualization. (c) Stanford bunny rendered as cloud with resolution 0.02.

ABSTRACT

Volume rendering has always been a popular topic in computer graphics as it captures more realistic rendering results. However, most of the rendering targets are focused on participating media distributed in the scene. In this project we integrate an Octree data structure with our path tracer in CUDA to achieve efficient volume rendering of models.

Our efforts are put on the implementation of a compact model representation in Octree with an efficient parametric algorithm for tree traversal. With the proposed Octree combined into our CUDA path tracer, the volume rendering by each ray per thread is made possible for average Nvidia graphics cards. Our rendering algorithm extends the classic ray tracing process into path tracing with light transport in participating media, which gathers light energy in a physically based manner. Finally, the global illumination examples and comparisons of cloud like models are shown.

Keywords:
Octree, Volume Rendering, Participating Media, Path Tracing, CUDA, Global Illumination, Acceleration Techniques

1 INTRODUCTION

Rendering of participating media provides realistic and natural effects, such as cloud, fog, smoke and steam, in real life. With volume rendering effects, the resultant images/videos often become more visually appealing as the volumetric effects adds mood into the scene.

For high quality rendering, several Monte Carlo and finite element techniques have been proposed. These methods can model general volumetric phenomena and scattering effects. Unfortunately, the full Monte Carlo gathering process may be extremely expensively costly in the traditional rendering pipeline. As multi-cores CPU and general purpose graphics processing units become available to average consumers, the full gathering of traditional, indirect path tracing becomes possible to be realized in software based graphics rendering pipeline, such as in CUDA, OpenCL…etc. However, the computation of physically based volume rendering is still expensive which may require several hours so as to render a simple model or simplified volume data representation. In this project, we adopt a compact Octree data structure as the representation of models in scenes and integrate it into our CUDA path tracer for efficient volume rendering. The exploitation of dedicated graphics hardware is shown and the results are realistic and visually appealing.

The remainder of the report is organized as follows: We start from introducing related work and relevant algorithms in section 2. In section 3, we give an overview of an efficient parametric algorithm for Octree Traversal. The path tracing algorithm is briefly reviewed in section 4, and

we describe how to integrate the volume model in Octree representation into a path tracer. In section 5, we propose the CUDA implementation of the whole algorithm and show the results, and in section 6 we discuss the method and ideas for improvement. Lastly, in section 7 we close with a conclusion.

2 RELATED WORK

Numerous efforts on simulating volume effects can be traced from Blin's paper [17]; due to the scope of this project, we only discuss important representative papers here. Monte Carlo ray tracing methods were brought to computer graphics by Cook at el.[11] to render realistic effects including multiple scattering and non-homogeneous media. With the methods of compositing images by Porter and Duff[10], the volume rendering can be viewed as a problem of composing colors of finite elements in a volume by ray tracing algorithm as Levoy and Drebin et al. proposed[2, 3, 4].

Among others, one research direction has led to volume rendering techniques that exploit hardware assisted texture mapping. These methods represent volume data as 3D texture and further render the target cells by texture look up. Modern graphics hardware enables this volume rendering approach to be performed in real-time. This type of volume rendering was first described by Cullip and Neumann[12], and was enhanced and extended into the OpenGL's shading language by Westermann and Ertl[6] or even into general purpose GPU as Tatarinov and Kharlamov presented [6]. We hope to capture volume effects in a physically based ray tracing manner in this project; therefore, we do not adopt rendering volume data explicitly by using graphics hardware (3D texture look up).

Kajiya[86] proposed the famous rendering equation for gathering lights in an integration manner. Many contributions were done based on this theory. Accompanied with distributed ray tracing algorithm [11], an enormous amount of research on physically-based rendering has been proposed. Lafortune at el.[5] render participating media with bidirectional path tracing, in which the involvement of the participating media rendering is also a probability distribution function. Recently, Kulla and Fajardo[14] add importance sampling techniques into their path tracing process for the purpose of lower variance and faster convergence. On the other hand, Jensen and Christensen [13] combine 3D photon mapping with participating media for more efficient simulation of light transport. Sun at el.[16] use an analytic method to turn the heavy volume rendering integration equation into a simple but precise function without loss of general properties of participating media. With the knowledge of the aforementioned papers above, our goal in this project is to view the volume rendering problem as a color compositing process by transporting lights in a compact volume representation.

Revelles, Urena and Lastra[9] compare and propose a new method of Octree traversal. Details of their parametric algorithm and performance comparisons are described in their paper. In our best knowledge, its simplicity and low computation complexity makes it a fairly robust Octree traversal algorithm, which attracts us to implement and integrate into our path tracer. We start from briefly describing their Octree traversal algorithm.

3 PARAMETRIC OCTREE TRAVERSAL

3.1 OCTREE DIFINITION

Octree Traversal is the process of finding the subset of voxels in an octree pierced by a directed line. In [9], the authors introduce a new top-down algorithm which is based on the parametric representation of the ray (line). As described below, the simplicity of the traversal algorithm makes it straightforward to implement and provides a light-weight package for us to integrate the algorithm into our path tracer. Figure 1 shows the octree labeling in the algorithm, which we use for our octree traversal.

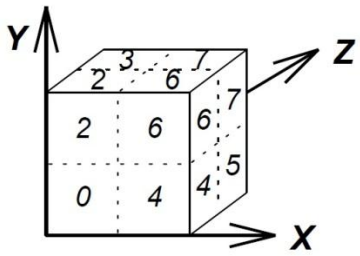


Figure 1: Labeled octree (the hidden node has label 1).

3.2 THE ALGORITHM FOR 2D CASE

We define a ray *r* as a pair *(p, d)*, where $p = (p_x, p_y)$ is the origin, and $d = (d_x, d_y)$ is the unit length direction vector.

From the above definitions, we deduce that an intersection between a ray *r* and a node *o* (a rectangle on 2D plane) occurs if at least one real value *t* exists such that:

$$x_0(o) \leq x_r(t) < x_1(o) \quad \text{AND}$$
$$y_0(o) \leq y_r(t) < y_1(o) \qquad (1)$$

where:

x_0 is the minimum boundary in x axis

x_1 is the maximum boundary in x axis

y_0 is the minimum boundary in y axis

y_1 is the maximum boundary in y axis

The algorithm is called a *parametric* algorithm because all computations use the value of *t* such that $(x_r(t), y_r(t))$ is a point on a node boundary. For a node *o* and ray *r*, $t_{x0}(o,r)$, $t_{y0}(o,r)$, and $t_{x1}(o,r)$, $t_{y1}(o,r)$ are defined as the ray

parameter values for which the ray intersects with the boundary of the node.

Then we can define t_{min} and t_{max} for a node *o* and a ray *r* as

$$t_{min}(o,r) = \max(t_{x0}(o,r), t_{y0}(o,r))$$

$$t_{max}(o,r) = \min(t_{x1}(o,r), t_{y1}(o,r)) \qquad (2)$$

If a *t* exists obeying (2), then $t_{min} \leq t < t_{max}$. The inverse implication also holds, thus equation (1) is equivalent to

$$t_{min}(o,r) < t_{max}(o,r) \qquad (3)$$

If the condition of (3) is false, no intersection occurs. When the condition is true, all values of *t* in the interval $[t_{min}, t_{max})$ are mapped to points $(x_r(t), y(t))$ which belong to the node. It is now possible to outline the proposed parametric algorithm used to traverse a quadtree (since we are still discussing 2D case). First, we check condition (3) for the root node. If the condition is true, the four parameters (t_{x0}, t_{y0}) and (t_{x1}, t_{y1}) need to be computed for the root node by using line intersection equations. The main recursive procedure is subsequently executed accepting a node as input parameter and its corresponding four parameter values. If the node is non-terminal, its child nodes which are pierced by the ray are checked using (3) for each of them. Therefore, a recursive call to the procedure is carried out for each of them.

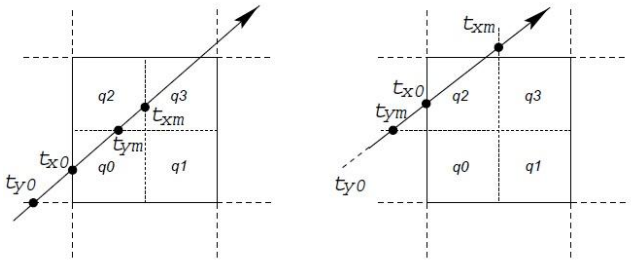


Figure 2: Sub-nodes crossed when $t_{x0} > t_{y0}$ (2D case).

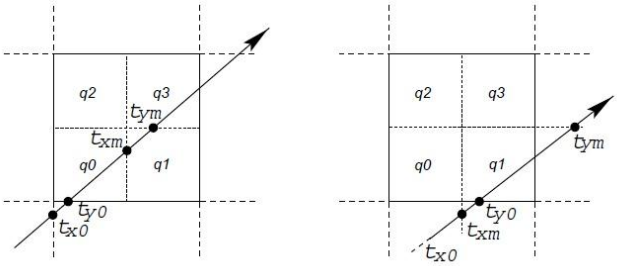


Figure 3: Sub-nodes crossed when $t_{y0} > t_{x0}$ (2D case).

In fact, there are only six different parameters for a child node. These are the four parameters of the parent plus the following two values:

$$t_{xm}(o,r) = \left(t_{x0}(o,r) + t_{x1}(o,r)\right) / 2$$

$$t_{ym}(o,r) = \left(t_{y0}(o,r) + t_{y1}(o,r)\right) / 2 \qquad (4)$$

This kind of coherence can be used to improve the algorithm further by using a sequential algorithm. The selection of pierced sub-nodes of a node is carried out in two steps:

1. Select the first sub-node hit by the ray
2. For each pierced node, select the next one, until the current parent node is exited.

The two steps are recursively performed while the ray marches (see figure 2 and 3).

3.3 EXTENDING THE ALGORITHM TO OCTREES

To find the first sub-voxel at which the ray enters the current voxel, first we obtain the entry face of the current voxel. This step is made by computing $\max(t_{x0}(o), t_{y0}(o), t_{z0}(o))$. In table 1 we show the entry plane selected for each case.

Maximum	Entry plane
t_{x0}	YZ
t_{y0}	XZ
t_{z0}	XY

Table 1: First plane intersected.

Once the entry plane has been determined, four sub-nodes are candidates (see figure 1 for the octree labels). To determine the first sub-node crossed, we examine $t_{xm}(o)$, $t_{ym}(o)$, and $t_{zm}(o)$. In table 2 the necessary comparisons are shown. The result of evaluating this condition (a bit) is copied to one of the bits which form the index of the first sub-node crossed. In this way, any node could be selected with the exception of node 7, because the ray direction vector components are assumed to be positive (negative case is described later). The whole process can be implemented using the *OR* operator to combine the necessary bits.

Entry Plane	Conditions to examine	Bit affected
XY	$t_{xm}(o) < t_{z0}(o)$	0
	$t_{ym}(o) < t_{z0}(o)$	1
XZ	$t_{xm}(o) < t_{y0}(o)$	0
	$t_{zm}(o) < t_{y0}(o)$	2
YZ	$t_{ym}(o) < t_{x0}(o)$	1
	$t_{zm}(o) < t_{x0}(o)$	2

Table 2: Comparisons to obtain the first node intersected.

The same procedure is carried out recursively until the ray exits the root node. As to the negative ray direction, one can imagine we address the problem by transforming the whole octree orientation so the direction of the ray is always positive. There are some extreme cases of ray paralleling to the x, y, or z axis, we can simply shift the ray direction by a small value, i.e. epsilon 1e-10, to prevent any numerical error or singularity. For whole details of the ray marching in the octree, we suggest viewing the paper [9] directly.

4 VOLUME RENDERING IN PATH TRACER

4.1 VOXLE COLOR MODEL

With the compact octree structure and traversal algorithm under the belt, we can start to introduce our lighting model of the voxel data. As we mentioned above, our goal is to turn the volume rendering problem into a case of compositing colors. Following [3, 4], we obtain the equation for compositing front/back colors:

$$C'_{front} = C_{front} + \left(1 - \alpha_{front}\right) * C_{back} \qquad (5)$$

where C'_{front} is the composited color of viewing the front voxel, C_{front} is the original color of the front voxel, C_{back} is the color of the back voxel and α_{front} represents the alpha channel of the front voxel. Notice that we use the same equation (5) universally in our implementation of rendering leaf voxels in the volume octree. Figure 4 shows an overview of the volume rendering algorithm.

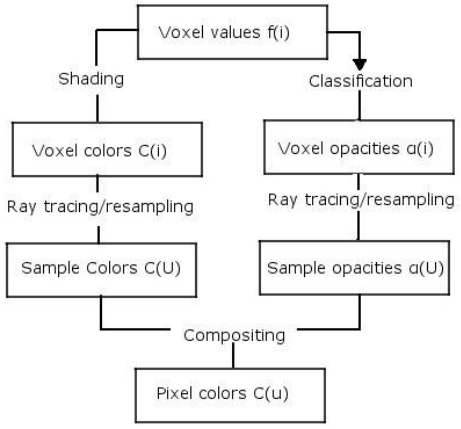


Figure 4: Overview of volume rendering algorithm.

Given a ray marching through the scene, we check if it intersects with the volume octree. If not, we simply continue the light transport algorithm and gathering lights. If it does, we collect voxel colors as the ray pierces through those voxels and use equation (5) to compose them together into the current pixel which the ray represents. Whenever the light exits the octree (if intersection happens), we need to attenuate the ray's throughput so that the subsequent light transport procedure abides by the energy absorption of volume. For 2D illustration of the ray marching through volume process, see Figure 5.

Until now we assume we have already known the color of the voxels before we start compositing them, which is incorrect since those voxel colors are supposed to be determined by their relation with light sources, instead of using pre-determined voxel colors, e.g. from CT data. Therefore, for those volume data, we must launch another light transport procedure from each ray-intersected voxel to the light source, just like shooting a new shadow ray when calculating diffuse light component.

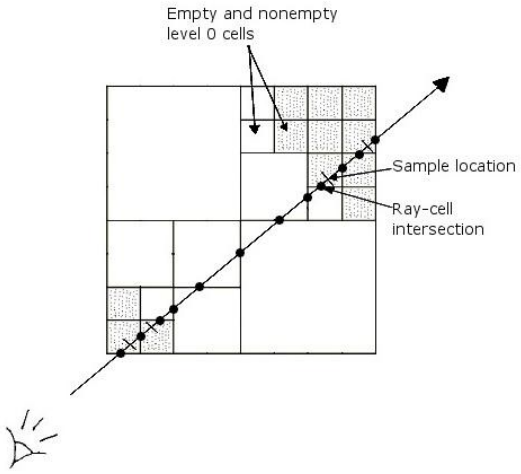


Figure 5: Ray marching through an octree node.

4.2 GATHERING VOXEL COLOR

Following equation (5), we know the final compositing color of a ray through *n* voxels in the volume should be

$$C_{final} = C_0 + (1 - \alpha_0) * C_1 + (1 - \alpha_0)(1 - \alpha_1) * C_2 + \cdots + (1-\alpha_0)(1 - \alpha_1) \ldots (1 - \alpha_{n-1})C_n \qquad (6)$$

If we assume volume data are isotropic and homogeneous, i.e. $\alpha_0 = \alpha_1 = \cdots = \alpha_n$, then (6) can be written as

$$C_{final} = C_0 + T^1 * C_1 + T^2 * C_2 + \cdots + T^n C_n \qquad (7)$$

where $T = (1 - \alpha)$

As mentioned in previous section, for each voxel color C_i, we need to launch another light transport process from the position of voxel *i* to each of the light sources to determine how much irradiance the voxel catches. Figure 6 illustrates the idea of gathering light for each voxel.

The new gathering procedure only needs some modification from equation (6). We can view the new gathering ray as a way to determine how much energy a ray from light source arrives at voxel *i*. In other words, we calculate the final light energy from light source through finite volume absorptions as the ray goes and eventually survives at voxel *i*. Furthermore, the color compositing equation (5) can be interpreted as the approximation of light energy not being

absorbed, in which $(1-\alpha)$ represents the remaining light energy after passing through a single voxel. Therefore, for each voxel i, its final color C'_i after gathering lights is

$$C'_i = C_i\left(1-\alpha_0^i\right)\left(1-\alpha_1^i\right)\ldots(1-\alpha_m^i)L \quad (8)$$

where C_i is the unshaded voxel color, α_1^i represents the alpha channel of voxel j and L means the light radiance. We can rewrite the equation (8) by assuming the volume data are isotropic and homogeneous as before:

$$C'_i = C_i T^{m-1} L \quad (9)$$

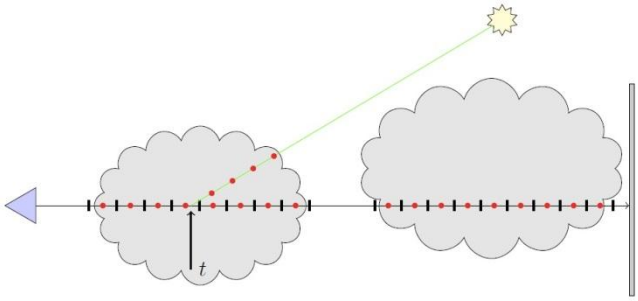


Figure 6: Gathering voxel color from light source.

4.3 INTEGRATION INTO PATH TRACER

Our path tracer runs the classic rendering equation [1]:

$$I(x,x') = g(x,x')[L_e(x,x') + \int_S \rho(x,x',x")I(x',x")dx"] \quad (11)$$

where:

$I(x,x')$	is the related to the intensity of light passing from point x' to point x
$g(x,x')$	is the geometry term indicating the geometry relation of x and x', often a distance falloff
$L_e(x,x')$	is related to the intensity of emitted light from x' to x
$\rho(x,x',x")$	is related to the intensity of light scattered from x'' to x by a patch of surface at x'

Average rays gather color (light energy) by following the above equation if they do not intersect with the volume octree. If the intersection happens, we use the refined version of equation (11):

$$I(x,x') = g(x,v)[C_v] + T^{n+1}g(x,x')[L_e(x,x') + \int_S \rho(x,x',x")I(x',x")dx"] \quad (12)$$

where:

v	is closest voxel reached from the current ray
n	is the number of voxels the ray passes in the octree volume
$g(x,v)$	is the geometry term indicating the geometry relation of x and v, often a distance falloff
C_v	is the gathered voxel color from equation (7)
T^{n+1}	affects the throughput of the ray after passing through the volume

5 IMPLEMENTATION

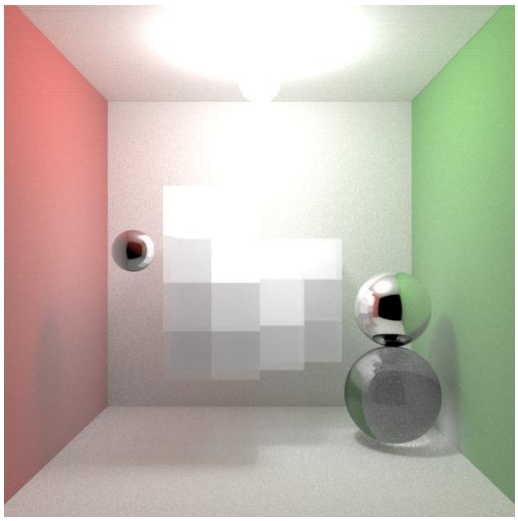
5.1 SOFTWARE AND HARDWARE SPECIFICATIONS

Our path tracer is implemented in CUDA 4.0. We use the following hardware set: GeForce GTX 770 Nvidia graphics card and Intel i7-4770k CPU for all the rendering results in this report.

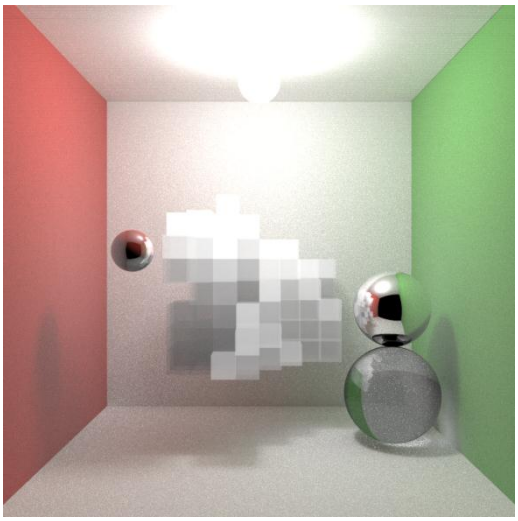
64 threads are running in the same block on the device. The path tracer is implemented with stream compaction on rays, Russian Roulette techniques for early ray termination to enhance the performance and shorter converge time. Each ray is given 5 bounces at most for transporting in the scene.

As we integrated the octree into our path tracer, the issue of copying memory from host side (DRAM) to device side (global memory on graphics cards) came up because currently there is no API provided for copying dynamically allocated memory from host side to device side in our knowledge. Therefore, it is necessary to turn our octree representation (octree is a linked list of an unbalanced tree structure) into an array of our new defined octree node structure programmatically. Each octree node stores the node's location in 3-space, a flag bit indicating if it is a leaf node in the octree, a flag bit showing if the leaf node has data or not, and finally the node has to record the unsigned int array indices of its eight children. Therefore, the total size per node is 44 bytes. For most cases, rendering a volume model with octree resolution ~0.03 produces pleasant results, and it requires about 10,000 nodes per octree, which only needs an array of size ~440 KB to be sent to CUDA memory. It is possible to reduce the memory consumption further, such as by using index array, similar to IBOs in OpenGL, instead of storing the location for each node directly (because the positions of non-leaf nodes are never used in the algorithm); however, we did not implement this technique since we did not meet any memory overflow issue in all of our test sets.

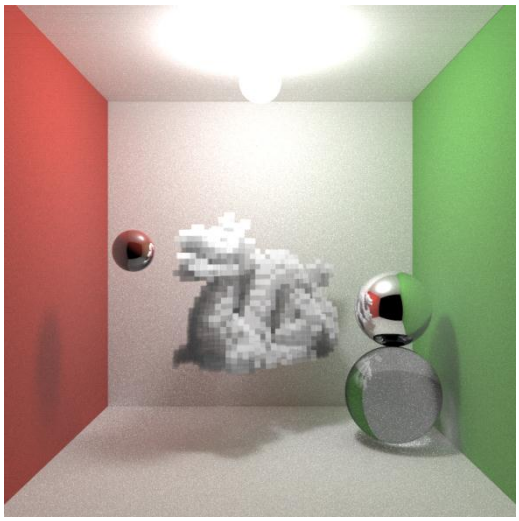
Since CUDA 2.0, recursion calls are allowed in kernel functions. Our octree traversal algorithm, based on [9], is implemented with recursive functions. However, if we perform the full recursion function for the octree traversal as described in section 4.1 and 4.2, there would be a recursion inside another recursion, which is risky for any thread call in CUDA. In order to prevent nested recursions, we adopt a two passes method to gather to light energy. In the first pass, we simply record the positions of the voxels that a ray reaches. Then in the second pass, we shoot a new ray from those pre-recorded voxel locations to each of the



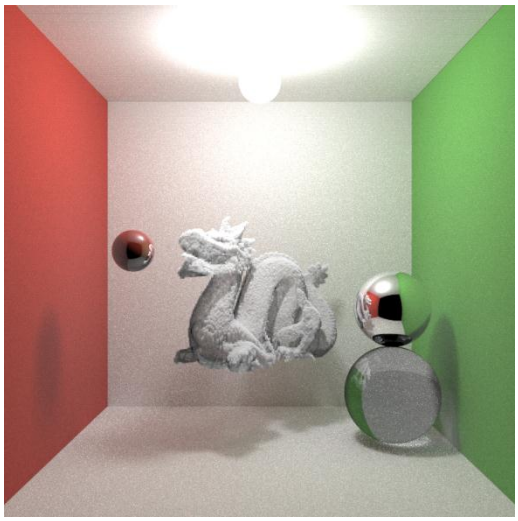
a



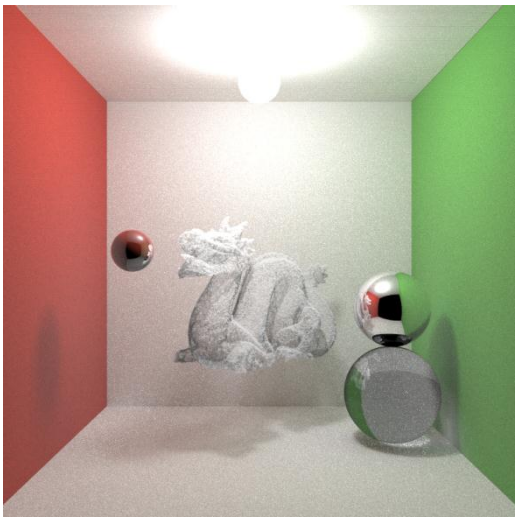
b



c



d



e

	Octree resolution	Rendering time
a	1.00	5m 18.561s
b	0.30	9m 27.832s
c	0.10	21m 5.2s
d	0.03	33m 0.62s
e	0.015	33m 35.06s

m: minute(s)

s: second(s)

Figure 7: Stanford dragon rendered as cloud with different octree resolutions. Notice how the opacity of the dragon Changes w.r.t. different octree resolutions. That is caused by equation (7) and the transmittance threshold.

light sources to gather light energy by following equation (9). After the gathering process for each voxel in the second pass, we evaluate the composited color by equation (7) and continue our light transport process in path tracing.

We also set up a transmittance threshold for early termination of over dim ray when it pierces too deep in the volume. This technique can effectively stop a ray if it contributes only an extremely small amount of energy to the pixel.

5.2 RENDERING DETAILS

We set the alpha channel of all test sets to be [0.3, 0.4], and find the alpha in the range makes no noticeable difference to human eyes. The theoretical execution time is proportional to the square of the inverse of the octree resolution since we need to examine and gather light energy at each voxel location. However, since we have early termination mechanism, this time is only proportional to the inverse of the octree resolution in practice, and is often less than that. Figure 7 shows the volume octree representations of the same model (Stanford dragon) with different octree resolutions and running times. Experiment on how many Monte Carlo iterations we need for the noise removal of the volume rendering is taken as well, see Figure 8 for visual details. One can see that the cloud effects of the bunny by running 1,000 iteration and the one by running 5,000 iterations are similar since the volume rendering procedure does not involve in diffuse component calculation, which is often the main cause of high variance in Monte Carlo method.

The diffuse component in the path tracer is using Lambertian reflectance model; the reflection and the refraction reflectance is followed by Fresnel equation with Russian Roulette to decide whether to reflect or refract. The light source is simulated as a point light; therefore, we lift the model off the ground and farther from camera so as to have a better visualization of the volume rendering effects. We also sample the point light by sampling the actual sphere surface of the point light with respect to its radius for realistic shading effects.

6 DISCUSSION

While our algorithm in volume rendering is straightforward, the use of color composition for participating media is an approximation of natural effects. Several improvements can be achieved for more precise physical results. We refer the famous brightness function in volume rendering from [17] and infer we only need the following factor in their volume brightness function (see Appendix section for the notations and the details of the equation):

$$\varphi(a) \qquad \text{and} \qquad \int_0^T e^{-nV} dT'$$

Since we are gathering light energy from light transports, we can omit most parts of the equation because they are implicitly included in the path tracing algorithm and keep only the above two terms.

Therefore, by adding a phase function $\varphi(a)$ to our lighting equation in octree traversal, we are able to simulate anisotropic scattering volume. By adding the integration $\int_0^T e^{-nV} dT'$, the precise scattering and transparency of particles can be easily obtained. Notice how we can get the information of n and V without extra effort, since they are included in the octree structure. In fact,

$$e^{-1} = 0.367879 \ldots$$

which is fairly close to our opacity range [0.3, 0.4] and we also use exponential fade-off (e.g. T^i term in (7)) in our scattering model. Also, in isotropic media the phase function $\varphi(a)$ is a constant $1/4\pi$. This explains how our isotropic volume approximation is close to the physical results.

Another potential improvement will be the compression in the octree structure sent to CUDA memory because of the existence of multiple node duplications. Recently, [15] provides an intuitive and effective way to compress octree structure losslessly by using directed acyclic graphs, which can reach a compression ratio of 38 on average. With this technique the memory consumption in CUDA can be reduced immensely.

A potential limitation of our algorithm is the rendering reliance on graphics hardware because the reading time in CUDA's global memory is much larger than in CUDA's shared memory and thread registers. Therefore, the overall rendering process can be performed more efficiently if we exploit shared memory in CUDA. On the other hand, host side memory and device side memory swapping time can be reduced by calling *cudaHostAlloc();* funtion in CUDA. The function allocates a fixed memory on host side that can be accessed from device side. However, since we assume our scene to be simple and static, it does not affect the performance in our rendering set. This functionality can be a potential use if we want to add dynamic objects into the scene.

Multi-scattering is the main limitation in our algorithm since our path tracing process only handles a single ray per thread for efficient rendering. One can add a *pdf* (probability density function) of multi-scattering to the shading equation if the scattering function is given. In fact, Veverka [18] points out that the multiple scattering law should provide a physical basis for the purely empirical Lambert's law, which can be easily added to our shading model with a few other rays sampling the surrounding of the volume. We view it as a potential method for multi-scattering.

7 CONCLUSION

We propose a new viewpoint of volume rendering in this project as reducing the problem to color composition in 3-space. A parametric octree traversal algorithm is implemented into our path tracing pipeline for efficiently classifying ray-volume intersections. The modern graphics hardware enables the Monte Carlo path tracing practically in our physically based rendering manner. Different time consumptions with respect to each octree resolution are provided. Comparisons of results of several Monte Carlo iterations and octree resolutions are also shown. Furthermore, the volume rendering implementation in CUDA is described and only requires a fairly small amount of memory for our octree volume representation. In sum, we meet our goal in this project of efficient volume rendering.

ACKNOWLEDGMENTS

We wish to thank Professor Ulrich Neumann for teaching, sharing knowledge with us on this course (CSCI 580 at USC) and giving us advice when we proposed our idea about the project. Also we appreciate the dedication of the TAs for helping and solving problems in the course realm. The dragon and bunny models are downloaded from Stanford 3D Scanning Repository:

http://graphics.stanford.edu/data/3Dscanrep/

REFERENCES

1. James T. Kajiya "The Rendering Equation" in SIGGRAPH '86
2. Marc Levoy "Display of surfaces from volume data", University of North Carolina at Chapel Hill, 1988
3. Robert A. Drebin, Loren Carpenter, Pat Hanrahan "Volume Rendering" in SIGGRAPH '88 Page 65-74
4. Marc Levoy "Efficient ray tracing of volume data" in ACM Transaction on Graphics, Vol. 9 Issue 3, July 1990

5. Eric P. Lafortune, Yves D. Willems "Rendering Participating Media with Bidirectional Path Tracing" in Eurographics Rendering Workshop, 1996
6. Rudiger Westermann, Thomas Ertl "Efficiently using graphics hardware in volume rendering applications", in Proceeding SIGGRAPH '98 Pages 169-177
7. Tatarinov, Kharlamov "Alternative Rendering Pipeline Presentation", NVIDIA SIGGRAPH 2009: http://goo.gl/bXhMe0
8. Matt Pharr, Greg Humphreys "Physically Based Rendering": http://www.pbrt.org/
9. J. Revelles, C. Urena, M. Lastra "An Efficient Parametric Algorithm for Octree Traversal", WSCG '00
10. T. Porter, T. Duff "Compositing digital images", in Proceeding SIGGRAPH '84 Pages 253-259
11. Rober L. Cook, T. Porter, L. Carpenter "Distributed ray tracing", in Proceeding SIGGRAPH '84 Pages 137-145
12. Timothy J. Cullip, Ulrich Neumann "Accelerating Volume Reconstruction with 3D Texture Hardware", UNC at Chapel Hill, 1994
13. Henrik W. Jensen, Per H. Christensen "Efficient simulation of light transport in scenes with participating media using photon maps", in Proceeding SIGGRAPH '98 Pages 311-320
14. C. Kulla, M. Fajardo "Importance Sampling Techniques for Path Tracing in Participating Media", in Journal Computer Graphics Forum, Vol. 31, Issue 4, June 2012 Pages 1519-1528
15. V. Kampe, E. Sintorn, U. Assarsson "High resolution sparse voxel DAGs", in ACM Transaction on Graphics(TOG) – SIGGRAPH '13, Vol. 32, Issue 4, July 2013, Article No. 101
16. B. Sun, R. Ramamoorthi, Srinivasa G. Narasimhan, Shree K. Nayar "A practical analytic single scattering model for real time rendering", in Proceeding SIGGRAPH '05 Pages 1040-1049
17. James F. Blinn "Light reflection functions for simulation of clouds and dusty surfaces", in Proceeding SIGGARPH '82, Pages 21-29
18. Veverka, J. "The physical meaning of phase coefficients", NASA SP-267, Pages 79-90

APPENDIX

A.1 VOLUME BRIGHTNESS EQUATION IN [17]

Once we have the octree representation of the volume, a volume brightness function can be written down:

$$B = (\omega/\mu)\varphi(a)n\pi r^2 \int_0^T e^{-nV} dT'$$

where:

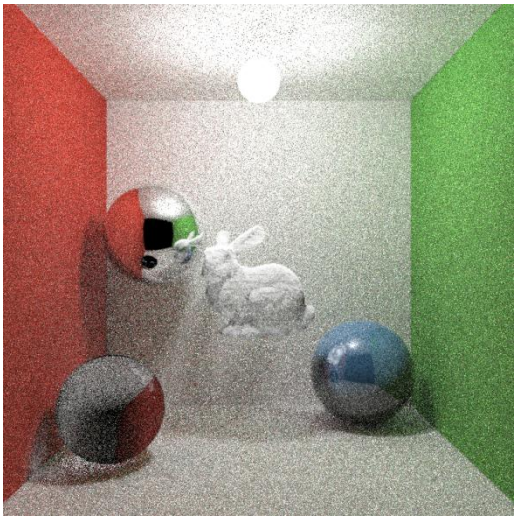
B	is the net brightness of the volume
r	is the radius of each particle in the volume
a	is the incident light angle
μ	is the cosine value of angle between viewing and particle normal
ω	is the albedo of individual particles
n	is the number of particles a ray intersects
$\varphi(a)$	represents a phase function, depending on the incident light angle a
V	is the path volume a ray goes by
T	is the ray path in the volume

From the view of physics, this equation means a ray goes into the volume from an angle a, meets n particles, and leaves the volume at $\cos^{-1}\mu$ direction. As the ray goes in the volume, the energy it brings is decreasing in a factor of e^{-nV}. Notice this is also a Poisson probability distribution of how many particles will be illuminated by the ray.

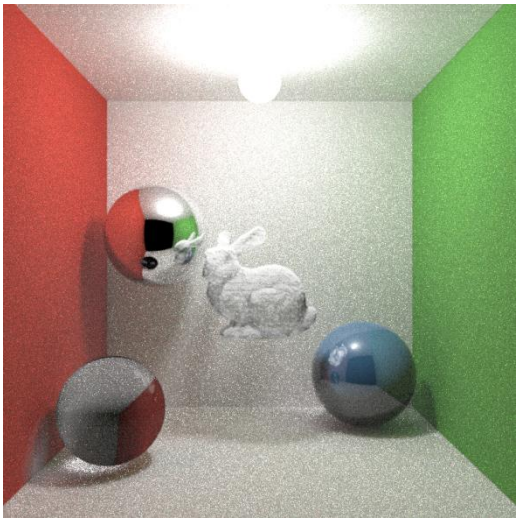
$(\pi r^2/\mu)$ can be interpreted as the projected viewing area.

$\omega\varphi(a)$ is the brightness of a particle with an light angle. ndT' is the expected number of particle/unit area

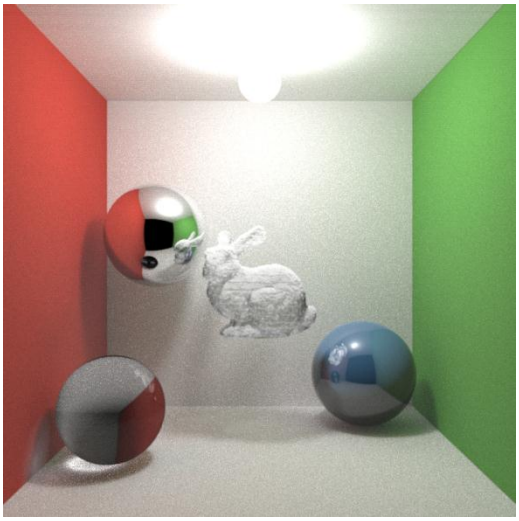
The tuning of density function in volume rendering is often added in the term V. Variance of e^{-nV} is also common, such as explicitly assigning scattering (σ_s) and absorption (σ_a) coefficients. This can be achieved by modifying e^{-nV} into a new Poisson probability model.



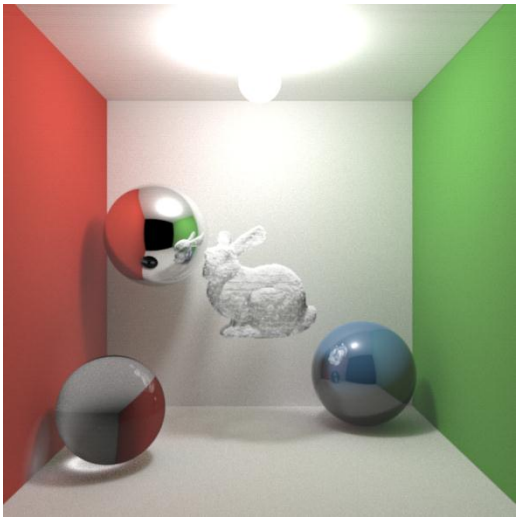
10 iterations, 8.939 s



100 iterations, 1m 22.844s



1,000 iterations, 13m 42.152s



5,000 iterations, 1h 8mins 34.15s

Figure 8: Rendering of Stanford bunny as cloud volume with different itrations.

h: hour(s), m: minute(s), s: second(s)