Volumetric Rendering of Participating Media

Alec Gemmell and Evan Maicus

Rensselaer PolyTechnic Institute
Troy, New York

Abstract
We present an extension of a standard ray tracer to support the rendering of participating volumes. These volumes, which may be either homogenous or heterogeneous, scatter and absorb radiance based on the Volume Rendering Equation. This results in visually pleasing simulation of smoke, fog, and other airborne media. In this paper, we discuss the volume rendering equation, the details of our implementation, and our algorithm's performance.

1 Introduction
In real scenes, all light not in vacuum travels through a particle containing medium. When attempting to model reality, then, it is natural to model the phenomena associated with these participating media. However, the number of particles in a medium is often on the order of tens of millions. As a result it is incredibly costly to model each of these particles individually. A much more computationally effective solution is to treat these particles as a **volume**, a region of space with properties determining the behavior of light within it via properties such as absorption, inscattering, emission and outscattering. With a scene partitioned into regions representing volumes, it is then possible to utilize the Volume Rendering Equation (described in section 3) to render a scene which includes participating volumes. In this paper, we present our implementation of an algorithm for use in a raytracer which renders such volumes.

2 Prior Work
When beginning to explore the topic of rendering participating media, the ideal starting point is likely "A Survey on Participating Media Rendering Techniques" [Cerezo, 2005]. As well as explaining the fundamentals of the topic area, "Survey on Participating Media" also enumerates commonly used phase functions (sometimes called transport equations), defines variables used in the domain, and explores methods of computing media effects both stochastically and deterministically. It was partly through our examination of "Survey of Participating Media" that we decided to implement a Monte Carlo ray marching approach.

A second paper we examined was "Radiance Caching for Participating Media" [Jensen, 2008]. This paper lead us to the the most influential work that we read, "Efficient Monte Carlo Methods for Light Transport in Scattering Media" [Jarosz, 2008]. "Efficient Methods" is a dissertation written by one of Jensen's students which encompasses

"Radiance Caching" while also providing a more nuanced explanation of its implementation. In its fourth chapter, "Efficient Methods" explores the mathematics of light transfer in a medium. These mathematical concepts, augmented by those from "A Survey of Participating Media" are examined in section 3 of this paper. In the fifth chapter of "Efficient Methods," Jarosz presents a novel method for computing illumination gradients while performing a marching Monte Carlo ray trace of a scene. In his implementation, Jarosz derived formulas for computing the gradients of the various terms in the Volume Rendering Equation, and was able to use these to approximate the radiance at a location using a set of already computed cached points. To extrapolate these values, Jarosz used an exponential function rather than a standard linear function or constant value. This resulted in much more accurate output than standard path tracing in the same or better time. While we did not implement Jarosz's caching due to time constraints, his dissertation provided the background that we needed to implement our Monte Carlo path tracer.

3 The Volume Rendering Equation

The key to rendering participating media is an understanding of how light interacts with a medium. This interaction can be fully described via the **Volume Rendering Equation,** also known as the Radiative Transfer Equation, described by Jaroz in his 2008 dissertation and Cerezo in his 2005 Survey. This section aims to describe and distill the mathematics described in chapters 4 and 5 of Jaroz's dissertation.

The Volume Rendering Equation:

$$(\vec{w} \cdot \nabla)\, L(x \to \vec{w}) = -\sigma_a(x)\, L(x \to \vec{w}) - \sigma_s(x) L(x \to \vec{w}) + \sigma_a(x) L_e(x \to \vec{w}) + \sigma_s(x) L_i(x \to \vec{w})$$

x is the starting point of a ray,
$\vec{w}$ is a direction,
$(\vec{w} \cdot \nabla)\, L(x \to \vec{w})$ represents the gradient of light passing through a medium starting at x in direction $\vec{w}$,
$\sigma_a(x)\, L(x \to \vec{w})$ represents absorbed radiance,
$\sigma_s(x)\, L(x \to \vec{w})$ represents the outscattered radiance,
$\sigma_a(x) L_e(x \to \vec{w})$ represents the emitted radiance,
$\sigma_s(x) L_i(x \to \vec{w})$ represents inscattered radiance.

The Volume Rendering Equation computes the gradient of light in a medium by taking the four types of light interaction into account: absorbance, outscattering, emittance, and inscattering. When integrated between two points x and x' in a scene, this equation can be used to provide the amount of light passing between them and through any volumes between.

3.1 The Extinction Coefficient

The **Absorbance** at a point in a volume can be described as the amount of radiance that is removed from a straight beam of light due to particles of a medium absorbing it. The

outscattering of a volume can be described as the amount of radiance that is removed from a straight beam of light due to it being scattered by particles in the medium. For the purpose of the Volume Rendering Equation, there is no difference between these two occurrences. The **extinction coefficient** in a scene can be defined as the amount of light lost at a point due to absorption and outscattering in a medium [Jarosz 2008]. Therefore, it can be written as:

$$\sigma_t(x) = \sigma_a(x) + \sigma_s(x)$$

Where both $\sigma_a(x)$ and $\sigma_s(x)$ are queryable functions that return the absorption and outscattering at a point x in a medium. The gradient net loss of light can then be written as:

$$(\vec{w} \cdot \nabla_t)\, L(x \rightarrow \vec{w}) = -\sigma_t(x)\, L(x \rightarrow \vec{w})$$

Using this equation, the **transmittance** ($T_r(x' \leftrightarrow x)$)**,** or percentage of light that can travel between two points x and x' in a scene, can be computed with the equation:

Transmittance

$$T_r(x' \leftrightarrow x) = e^{-\tau(x' \leftrightarrow x)}$$

$$\tau(x' \leftrightarrow x) = \int_0^d \sigma_t(x + t\vec{w})dt$$

Using this formula, the reduced surface radiance between the eyepoint and a point of intersection due to light interaction can be computed.

3.2 Scattering

The amount of light that is able to travel between two points in a medium is not solely affected by transmittance; light can be *added* to a medium via **emittance** and **inscattering.** Emittance can be easily defined as light added to a medium by the particle itself, as in the case of fire or sparks. We deemed emittance to be beyond the scope of our project. Inscattering can be defined as the opposite of outscattering. Simply put, if light is lost due to scattering between points x and x', it must have been picked up somewhere in between. For example, the fog in close proximity to a blue sphere is blue tinted, as it captures some of the light that had been headed towards the eyepoint.

Inscattered radiance can be broken into two distinct parts, Single Scattering and Multiple Scattering.

Single Scattering: As its name implies, single scattering represents light that is scattered exactly one time on its path from a surface to the eye. Single scattering can be computed using the equations below:

Single Scattering

$$L_s(x \rightarrow \vec{w}) = \int_A p(x, \vec{w'} \leftrightarrow w)\, L_r(x \leftarrow \vec{w})\; V(x \leftrightarrow x')H(x \leftarrow x')dx$$

A represents the surfaces in the scene
P represents the media's phase function.
L_r represents the radiance arriving at x from x'
V is a variable set to either 0 or 1 which denotes the visibility of x from x'
H represents a geometry term affecting light transfer.

The Phase Function
The phase function of a volume describes the way that light moves after being scattered by one of its particles. The simplest phase function is the isotropic phase function, which scatters light equally in all directions. This phase function is represented by the constant $1/4\pi$. In more robust applications, use of the Henyey-Greenstein Phase Function can be used to model anisotropic scattering [Cerezo, 2005].

$$L_r(x \leftarrow \vec{w}) = T_r(x \leftrightarrow x')L(x' \rightarrow x)$$

V can be computed via a ray cast from x to x' and is set to one if the ray makes contact and zero otherwise.

$$H(x \leftarrow x') = \frac{n \cdot \frac{\vec{x-x'}}{\|x-x'\|}}{\|x-x'\|^2}$$

Multiple Scattering: Multiple scattering represents light that undergoes multiple scatter events. Multiple scattering can be found using the following equation:

$$\nabla L_m(x \rightarrow \vec{w}) = \int_{\Omega 4\pi} \int_0^{\infty} p(x, \vec{w'} \leftrightarrow w)\, T_r(x \leftrightarrow x)\; \sigma_s(x')\, L_i(x' \rightarrow \vec{w'})\, V(x' \leftrightarrow x)dr'd\vec{w'}$$

Due to time constraints, our implementation does not take multiple scattering into account.

3.4 Useful Form

With these equations defined, they can now be combined into one approximate form for use in our algorithm. This approximate form can be solved via **Monte Carlo Integration,**

using **ray marching**. We will call this equation the **Monte Carlo Volume Rendering Equation:**

The Monte Carlo Volume Rendering Equation

$$L(x \leftarrow \vec{w}) \approx T_r(x \leftrightarrow x_s)L(x_s \rightarrow -\vec{w}) + (\sum_{t=0}^{S-1} T_r(x \leftrightarrow x_t)\sigma_s(x_t)L_i(x_t \rightarrow -\vec{w})\Delta_t)$$

Where S is the number of steps of length Δ_t taken by a ray.

$$L_s(x \rightarrow \vec{w}) \approx \frac{1}{N}\sum_{j=1}^{N}\frac{pL_rH}{pdf(x'_j)}$$

Where pdf(x'_j) is a point distribution function and N is a number of directional samples taken on a sphere at the current position.

$$L_m(x \rightarrow \vec{w}) \approx \frac{1}{N}\sum_{j=1}^{N}\frac{pT_r\sigma_sL_iV}{pdf(r'_j)pdf(\vec{w'_j})}$$

Where the first point distribution function samples the sphere around the current position and the second samples direction.

4 Algorithm

Using the Monte Carlo Volume Rendering Equation described in section 3.4, it is possible to render participating media via ray marching. The algorithm that we used is as follows:

for every pixel on the screen
- cast a ray, r, through that pixel
- determine the t at which it hits the nearest object or set it to max value if it hits none
- **Cast function:**
- **for** the number of steps necessary to reach t
  - Compute $T_r(x \leftrightarrow x_t)L(x_s \rightarrow -\vec{w})$ at r.start + t*r.direction
  - **for** each non-emissive sphere
    - **for** a number of random geometry samples
      - compute L_s by casting marching rays towards the sample
  - divide LS by the number of samples*the number of non-emissive spheres
  - Compute the contribution of this step.
  - Add the contribution of this step to the total inscattered contribution
- Determine c = the effects of shadows and reflections at this point
- Determine transmission rate $T_r(x \leftrightarrow x_s)L(x_s \rightarrow -\vec{w})$ at this point
- **return** (transmission rate * c) + inscattered contribution

4.1 Implementation Details

The implementation of the participating media ray tracer was written in C++ using the OpenMP API. The data structures used consist of three item vectors, a medium class, and

a sphere class. The sphere class is used for all objects in the scene whether they be lights or surfaces. Representation for the sphere consists of three item vectors representing the center point, surface color, and emitted color (used for lights). Additionally, it contains floats representing the radius, the square of the radius (precomputed and stored to accelerate intersection checks), and transparency and reflection coefficients. The medium class is used to store the properties of the homogeneous isotropic participating media the renderer aims to simulate. For a homogeneous isotropic substance, the phase function is a constant 4π and thus we store it as a float in the medium class. Another float represents the extinction coefficient which is a sum of both the absorption and outscattering coefficients for the medium. Finally, the medium class contains a RGB color value stored in a three item vector. This value is intended to represent the color of fog if the raytracer were to travel an infinite distance.

The majority of the computation time is spent in the *trace* function. The trace function takes in four arguments: a ray origin, ray direction, the vector of objects in the scene, and the ray depth, and returns a color value as a three item vector. Ray depth is a representation of how many reflection/refraction events have occurred since the initial ray was cast, zero being the initial camera ray and five being the final ray. The first computation performed is iteration over the list of spheres to find the distance to the cast ray's closest point of intersection *tnear*, which is set to infinity if none are found. Following this, the contribution of the inscattered light is calculated, as it is now known that the program need only march over the length of tnear to search for contributions. Thus, if the current ray is an initial camera ray (depth==0), the program begins the marching algorithm. The program marches in single unit steps in the direction of the cast ray until the distance reaches tnear. At each step the current fraction of the ray that is absorbed is calculated, and the remaining fraction is used as a coefficient to weight the contribution from inscattering. The inscattering contribution is obtained from a call to a function, *getInscattered*, passing in the current step location and the objects vector as arguments. If tnear is equal to infinity, the inscattered contribution is returned as the value of the pixel.

If a sphere was hit, standard ray tracing calculations for reflective, refractive, and diffuse materials are performed to obtain the reduced radiance contribution. Details that may be of interest are the epsilon value used (1e-4) and the contribution from light for diffuse surfaces being scaled by the extinction coefficient similarly to the marched rays. Finally, the transmission rate at tnear is calculated and used to scale the sum of the inscattered contribution and reduced radiance contribution. This value is then returned as the output.

Literature describes several methods of calculating the inscattering contribution, but this project's implementation only calculates single scattering [Cerezo, Jarosz]. As mentioned before, the *getInscattered* function takes in an origin point and the objects vector as arguments, and returns a color (the scaled inscattering contribution) as a three item

vector. For single scattering, there were several methods found when surveying prior work. One method involved monte carlo based ray tracing in random directions from the origin point, throwing out any rays that did not intersect a sphere, and repeatedly casting rays until the desired sample count is met [Jarosz, 2008]. While this method is accurate, sparse scenes with many missed rays will significantly increase runtimes. The method used in the implementation shoots a fixed number of rays at a random point on the surface of each sphere, and then takes the average contribution of all rays as the output. The major drawback of this method is that it has the potential of overweighing how much light would realistically reach an area. For example, if a small red sphere and large green sphere were equidistant from a point in opposite directions, more random light bounces should come from the green sphere than the red sphere. Using our method both spheres would provide equal light contributions, assuming they were about equally lit themselves. Additionally, if multiple spheres are in a line from the sample point, it causes the sample point to overweigh the first sphere in the line. However, we found these drawbacks were not producing significant negative visual results, and that in fact increasing the contributions from the spheres helped alleviate the lack of multiple scattering in our implementation.

The rest of the implementation is somewhat trivial. The program initializes by seeding the random number generator with the current time and creating the objects for the scene. The render function is then called, taking in the list of objects. The render function sets up a predetermined image with a fixed resolution, camera position, and field of view, and begins filling up three arrays with the RGB values of the pixels obtained from calls to the trace function. After this, the function writes these values to a ppm file, concluding the render process.

4.2 Parallelism

Both the ray tracing and participating media rendering lend themselves to many possible parallel optimizations. Ray tracing is implemented in parallel by calling the main render loop over an array using an openmp pragma. The pragma states that each thread can independently execute each iteration of the loop, and shares the pixel array and the vector of objects between all threads. This is known as image space parallelism, and provides speedup nearly proportional to the number of threads in theory. One drawback is that for eye rays that lead to complicated intersection checks, the thread responsible can get held up. Additionally, if the raytracer renders to a viewing window, the threads must wait for the graphics api to perform its job. This is avoided by just writing the raytraced values to an array and writing the array to a file.

This implementation has several areas of possible improvement in regards to parallel execution. For one, each call to traceray is performed on the same thread as the initial eye ray it was spawned from. Theoretically, each call is completely independent and since the pixel the call belongs to is already known the program could be refactored to completely parallelize ray tracing. Additionally, if multiple scattering were to be

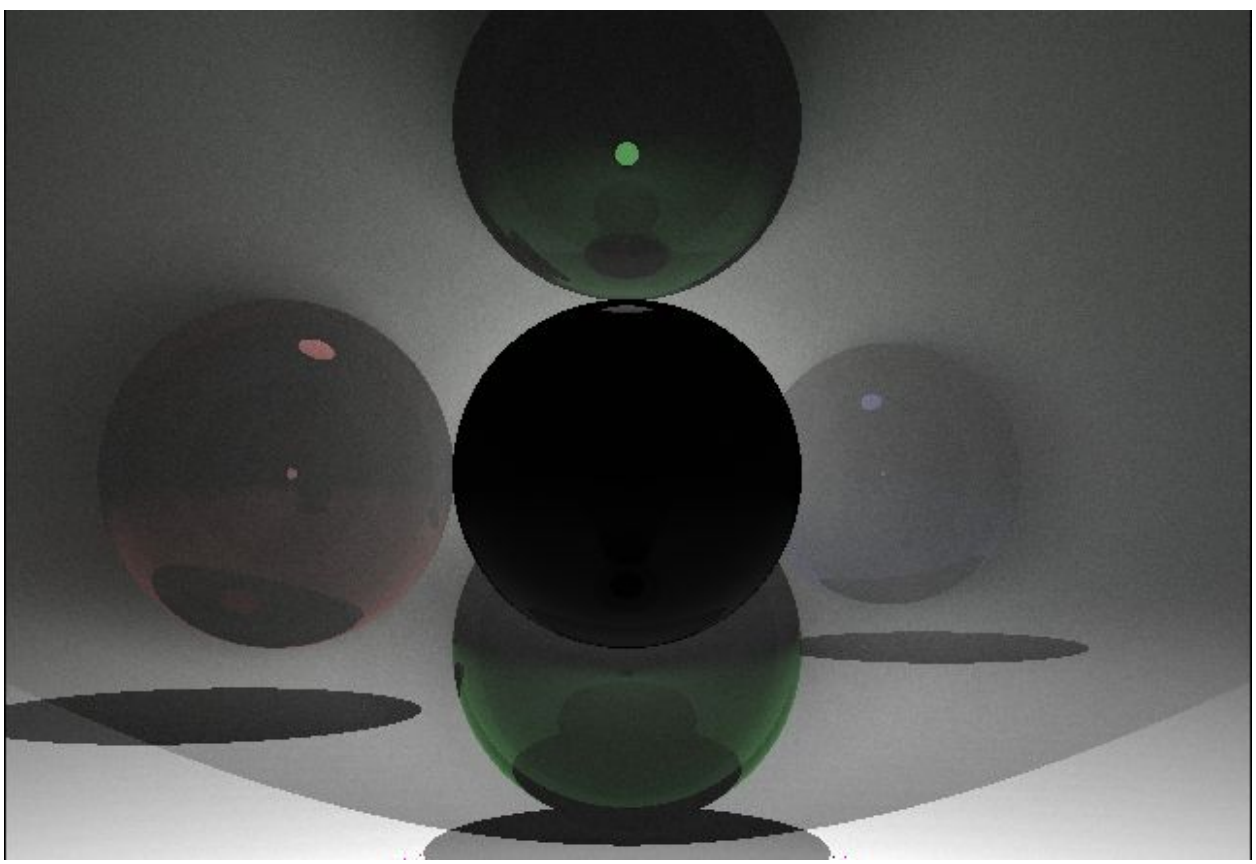
implemented with a radiance cache, the cache computation could also be implemented in parallel.

5 Results

Listed below are the relevant specifications of machines tested:

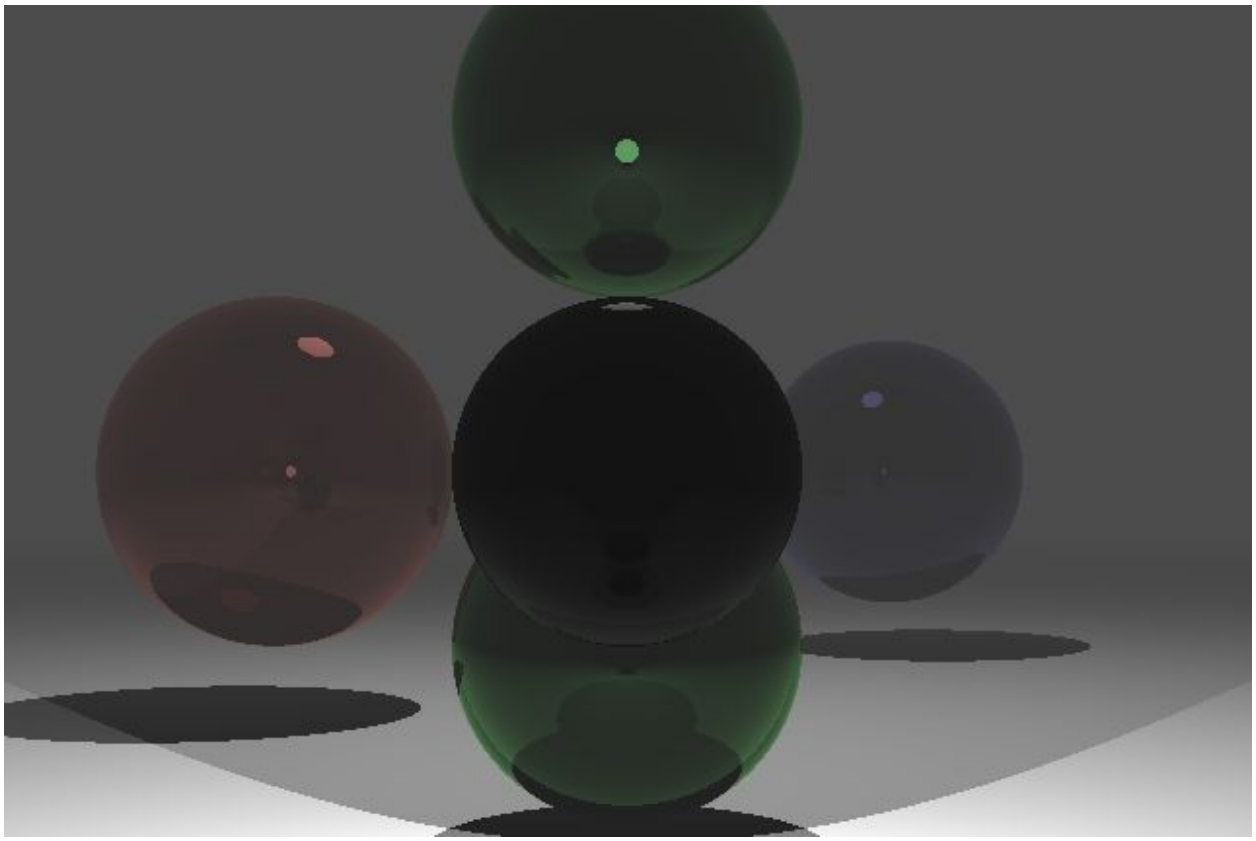
Desktop:
CPU: AMD Ryzen R7 1700 @ 3.4GHz (16 cores)
RAM: DDR4 16GB@2133MHz
ECSE Parallel Server:
CPU: Intel Xeon Phi 7120A @ 1.24GHz (61 cores)
RAM: DDR4 256GB@2133MHz

Results listed as no openmp were obtained by using the same source code with openmp pragmas commented out. The scene tested consists of a black sphere with a light source of the same radius directly behind it, completely obscured. Additionally a large sphere is used to represent the floor, and several other spheres are scattered about the scene. A secondary light source is above the scene out of view, indicated by the specular reflection on the red and blue spheres. Note the corona of light surrounding the black sphere created by the obscured light source in the render using inscattering.



Rendered image with 1 unit marching, 0.03 extinction coefficient, eight samples per sphere

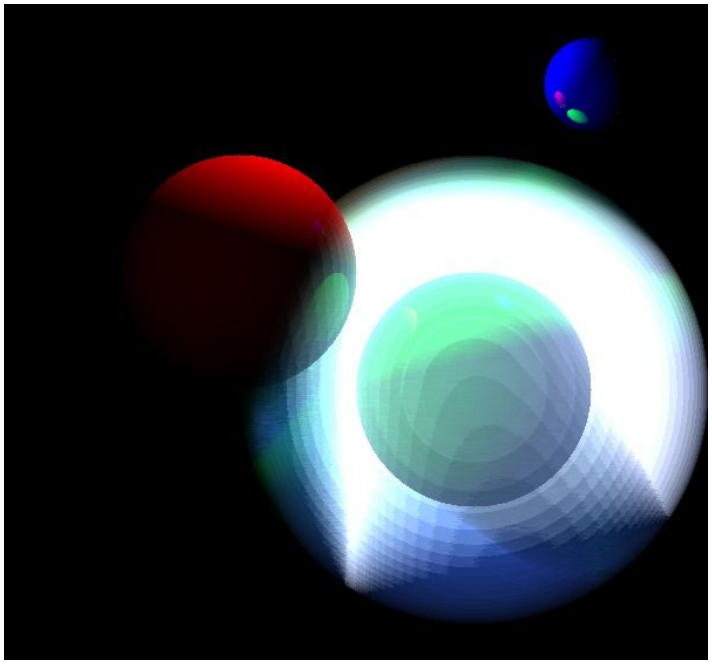
Results are listed as: system/samplecount/time(seconds)
desktop/8/406.987
desktop/4/208.643
desktop/2/101.919
desktop/1/52.1277
desktop/1/392.839 (no openmp)
desktop/0/0.536236
desktop/0/1.02807 (no openmp)
server/8/882.367
server/1/110.699
server/1/312.226 (no openmp)
server/0/0.139762
server/0/0.947051 (no openmp)



Flat color fog approximation for 0 samples, simulating reduced visibility but failing to diffuse light

Substantial speedup was gained from parallel optimization, up to 576% on the server and 653% on the desktop PC. The server likely suffered on the calculated fog runs due to the nature of the program not tracing the child rays in parallel. This means every new ray is traced on the same thread as the ray it is initially spawned in, and require the entire pixel to be calculated before the thread can move onto a new task. This creates

imbalanced loads and is partially alleviated in the desktop case due to the higher clock speed of the processor.



A Sphere of Heterogeneous fog with linear decay. The above image was created using an equivalent implementation of our algorithm using MPI in C

6 Conclusion

6.1 Individual Contributions

This assignment took us three weeks to implement. Weeks one and two were spent parallelizing and constructing ray tracers. Week three was spent solidifying our understanding of participating media, boiling down the mathematics into a usable format, and implementing Monte Carlo path tracing.

6.2 Future Work and Final Thoughts

Rendering participating media represents a difficult problem which many have tried to solve. Our implementation of a parallel Monte Carlo path tracer performs well and produces visually pleasing results and we are convinced that, given time, these results will only improve. Items that we would like to see implemented but did not have time to add include multiple-scattering, and an anisotropic phase function. We would also like to further explore heterogeneous media.

7 References

Cerezo, E., Seron, F., Sillion, F. X., Perez-Cazorla, F., & Pueyo, X. (2005). A Survey on Participating Media Rendering Techniques. Retrieved from http://eds.a.ebscohost.com.libproxy.rpi.edu/eds/detail/detail?vid=2&sid=42da1306-a821-4359-9c99-eeee650e80ad@sessionmgr4009&hid=4208&bdata=JnNpdGU9ZWRzLWxpdmUmc2NvcGU9c2l0ZQ==#AN=edsair.dedup.wf.001..7aa0c95933af7af69a5f9603e57461b2&db=edsair

Jarosz, W (2008). Efficient Monte Carlo Methods For Light Transport In Scattering Media Doctoral Thesis. Retrieved from http://search.proquest.com/openview/2687b024a8ef24087d8b20ef6feb178b/1?pq-origsite=gscholar&cbl=18750&diss=y

Jensen, H. W., Jarosz, W., & Donner, C., Zwicker, M. (2008). Radiance Caching for Participating Media. ACM Transactions On Graphics, 27(1), 1.