Volumetric Path Tracing

Justin Goodman
Dept. of Computer Science
University of Maryland
College Park, MD
jugoodma@umd.edu

***Abstract*—We implement volumetric rendering as an extension to the `nori` educational ray tracing codebase. We present renderings using both homogeneous and heterogeneous participating media.**

***Index Terms*—volume, volumetric, path tracing, participating media**

I. Introduction

Generating realistic images from a scene description is a common and well-studied problem in computer science. The general solution is to take a scene description, build a three-dimensional scene in-memory, and model light transportation (backwards) from the virtual camera eye into the scene. This is called ray tracing – where we shoot a ray from the camera eye into a virtual scene and compute the emitted light at the ray hit-point. This approach yields picture-realistic images, since it correctly approximates light transportation in the real world, at the expense of computational efficiency. Oftentimes, scenes are modeled using *meshes*, which may consist of millions of triangles. The naïve approach to compute a ray intersection is to test whether a ray intersects with *every* triangle in the scene. This is obviously computationally infeasible for a large number of triangles. To solve this, we store the scene mesh triangles in an *acceleration structure*. Examples include kd-trees, octrees, and bounding-volume hierarchies. Using this, we can logarithmically reduce the amount of triangles we have to intersect, thus making ray tracing computationally feasible. Finally, each individual section of rendered image is *independent* of all other sections, so we can easily parallelize the rendering computation.

This works for simple surface interactions. However, in reality, light often hits multiple surfaces before it reaches your eye. Each bounce alters the resulting light you see. To model this, we simply allow the light to bounce off a hit surface while ray tracing, and attenuate the final computed color per bounce. This process is called *path tracing*.

This generally gives us good results, and allows us to model scenes involving glass (caustics, specular highlights, etc) and mirrors. However, the actual path that light travels to hit your eye can alter the color as well. This is called a *volume* interaction, and we model this using *participating media*. Essentially, we consider tiny particles that light might travel through between surfaces all as potential hit points for the ray. We model this using simple absorption and scattering coefficients σ_a and σ_s. Particles also have densities, which may change throughout a scene, so we account for this by discretizing a scene indicating which areas have which densities of particles. This entire process is called *volumetric path tracing*, and is non-trivial to implement. We implement it in the educational ray tracer codebase `nori`, and provide resulting images rendered with a suite of participating media.

II. Related Work

James Kajiya created the rendering equation in 1986, which models light transportation and generalized many rendering algorithms at the time [1]. To-date, this equation is the main goal rendering algorithms aim to solve or approximate. Eric Lafortune applied monte carlo methods to solving the rendering equation in their 1995 thesis [2]. Then, a year later, Lafortune introduced volumetric path tracing to handle participating media [3]. Finally, `nori` is a modern ray tracer written in `c++`[1] by Wenzel Jakob, who also co-wrote, with Matt Pharr, the *Physically Based Rendering* textbook [4] and companion source code `pbrt-v3`[2]. We base our volumetric path integrator, and participating media interfaces, on those implemented in `pbrt`.

III. Approach

We extend the `nori` codebase with a volumetric path-tracing integrator. To do this, we implemented the following parts:

- `volpath.cpp` – the integrator itself
- `medium.h`, `medium.cpp` – interfaces for handling *medium interactions*
- `homogeneous.cpp`, `grid.cpp` – homogeneous and heterogeneous medium handlers
- `phase.h`, `henyeygreenstein.cpp` – interfaces for handling *phase functions* for medium interactions
- `scene.cpp` – for a special ray-intersect algorithm that accumulates scene transmittance in a medium
- `mesh.cpp` – to allow for meshes to have internal/external media [3]

[1] https://github.com/wjakob/nori

[2] https://github.com/mmp/pbrt-v3

[3] Unfortunately, I could not implement participating media *inside of* scene objects. I tried, but could not work out a bug that caused it to fail after 3 or 4 ray-medium bounces. I believe the bug deals with how I was switching the medium the ray is *currently in* while computing transmittance. This is not a huge concern, though, since one could quite easily use a grid medium with a homogeneous section located *inside of* the requested mesh object.

- `parser.cpp` – to account for inputting grid densities as an array

The simplest approach to volumetric path tracing is to assume the entire scene is inside a homogeneous fog. Fog is just a collection of particles that can scatter and absorb light. With standard ray tracing, where we only care about surface interactions, we shoot a ray into a scene and yield a hit-point on a surface. When introducing fog, we perform the same computation. However, there is *some chance* that our ray could hit a fog particle. Thus, we compute this chance by sampling the medium (the fog) with respect to the *transmittance*. For a homogeneous medium, this is computed as

$$\sigma_t = \sigma_a + \sigma_s > 0$$

where σ_a is the color absorbance coefficient and σ_s is the color scattering coefficient. We require the transmittance to be positive since we require non-negative color, and because we scale our sample probability by dividing the transmittance. To determine whether a ray interacts with the scene medium, we compute a distance $d = -\frac{\log(1-\epsilon)}{\sigma_t[c]}$ where we random uniformly select $\epsilon \in [0, 1]$ and we select, random uniformly, one of the three channel values c from σ_t. Then, we compute the time along the ray at which this selected medium interaction would occur, computed as $t = \frac{d}{||r||}$ where $||r||$ is the ray's direction length. If this time is less than the ray's maximum time, then we interpret this as a medium interaction. Otherwise, we interpret it as a surface interaction. We handle both interactions in a similar way – compute the contributed light using multiple importance sampling, then bounce the ray (using the phase function if medium interaction, or using the BSDF if surface interaction). Surface interactions are exactly the same as the `path_mis` integrator implemented in assignment 4. For medium interactions, the only difference is that shadow rays also have to include medium transmittance.

For the volumetric path integrator, our medium interfaces need to provide a sampling function to decide whether the current ray interacts with the medium, and they need to provide a transmittance function to compute how much light radiance is transmitted between two given points in the medium. For homogeneous media, by definition transmittance is constant everywhere, and is given by Beer's law

$$T_r = e^{-\sigma_t d}$$

where d is the distance between the two points. In our implementation, the two points are assumed to be the minimum and maximum points along the a given ray.

Our phase function interface requires a function to randomly sample a direction, as well as a function to return the probability that a given direction was sampled. We use the Henyey-Greenstein phase function model, which takes an asymmetry parameter $g \in (-1, 1)$ to control backward versus forward scattering based on the probability equation

$$\frac{1}{4\pi} \cdot \frac{1 - g^2}{(1 + g^2 + 2g(\cos\theta))^{3/2}}$$

given $\cos\theta$ between two direction vectors. Given these components, fog is quite easy to model (see fig. 1a). Similarly, we can also see the difference between backward and forward scattering (fig. 1b and 1c).

Though, this is only covers media where scattering is constant throughout the scene. Heterogeneous media have *non-uniform densities* in the scene. We account for this in the grid medium `grid.cpp` by storing densities as a *grid*. This is given to the ray tracer as an array of floating point values, as well as n_x, n_y, n_z parameters that dictate how the array is transformed into a three-dimensional matrix. We also require grid density media give a change-of-coordinate transformation to indicate how to change a world ray into a local grid ray. This is done by giving a grid density medium two local-coordinate points p_0 and p_1, as well as a transformation matrix W to go from local coordinates to world coordinates, and computing the inverse transformation. Using the homogeneous coordinate system, we get the matrix D that takes world coordinates to medium-local coordinates, given by

$$M = T(p_0) \times S(p_1 - p_0)$$
$$D = (W \times M)^{-1}$$

where T is the translation matrix

$$T(p) = \begin{bmatrix} 1 & 0 & 0 & p.x \\ 0 & 1 & 0 & p.y \\ 0 & 0 & 1 & p.z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and S is the scaling matrix

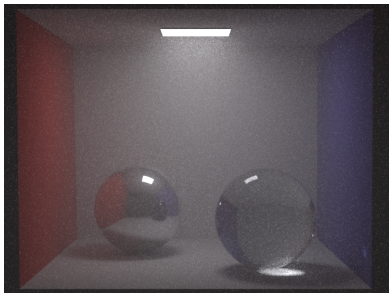
$$S(p) = \begin{bmatrix} p.x & 0 & 0 & 0 \\ 0 & p.y & 0 & 0 \\ 0 & 0 & p.z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

as usual. Then, we use D to translate rays to the grid medium local coordinate system for computing transmittance and sampling directions. The basic idea is we break the ray that is shot through the medium into individual homogeneous parts, and accumulate the transmittance according to Beer's law. This can be costly though, since we might have lots of three dimensional homogeneous sections in the medium. To get around this, one approach is to use *ray marching*, where we march through the heterogeneous medium along the ray in equidistant parts. This approach implicitly introduces bias though. Instead, we use *delta tracking*. To do this, we assume the heterogeneous medium is filled with "virtual" particles, and we compute transmittance (scaled by the maximum density) until we reach a uniform density by Russian roulette. But, we can think of the basic idea as advanced ray marching. An example of rendered smoke is shown in figure 2.

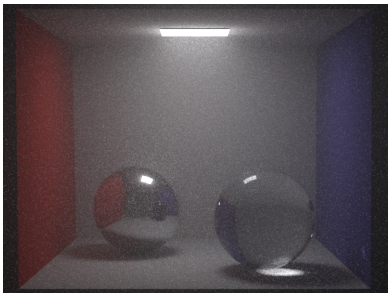
IV. Data

Using heterogeneous media, we can model things like clouds, smoke, and water. Smoke and cloud datasets are hard to come by.[4] To account for this, we wrote a small `Python`

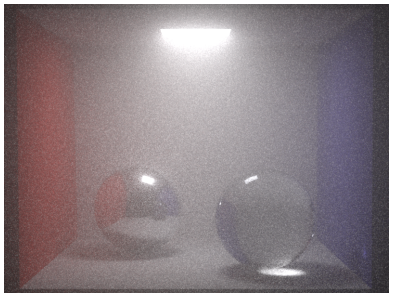
[4]We couldn't find any when we looked.



(a) Cornell Box rendered using volumetric path max-depth 5, 512 samples per pixel, $\sigma_a = [0.001764, 0.0032095, 0.0019617]$, $\sigma_s = [0.31845, 0.31324, 0.30147]$, and $g = 0$.



(b) Cornell Box rendered with the same settings, but $g = -0.9$. Backward scattering leads to light more likely scatting *towards* a light source. Thus, we see a darker image, since the camera eye is not *towards* the light source.



(c) Cornell Box rendered with the same settings, but $g = 0.9$. Forward scattering leads to light more likely scattering *away* from light sources. Thus, we see a brighter image, since the camera eye is *away* from the main light source with respect to the scene medium.

Fig. 1: Cornell Box rendered using varying values of g in the Henyey-Greenstein phase function.

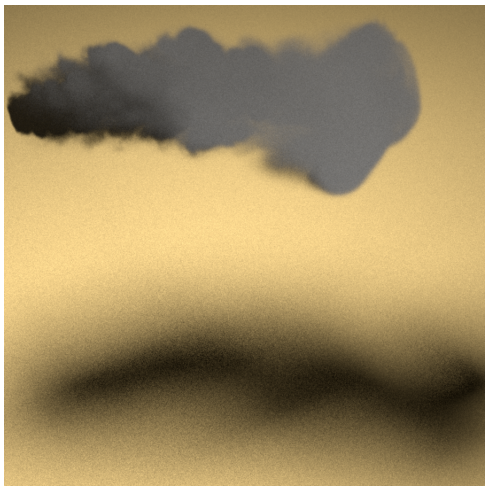


Fig. 2: Smoke using grid density values from the openly-available `pbrt-v3` scenes. This specific smoke cloud uses the *cloud* scene https://pbrt.org/scenes-v3. This is a different angle than the `pbrt` scene, but we can see the smoke blocks light and casts a shadow on the floor.

```
nx,ny,nz = 11,11,11
density = []
for i in range(nx*ny*nz):
  density.append(0)
for z in range(1,nz-1):
  for y in range(1,ny-1):
    for x in range(1,nx-1):
      dist =
        ( x - (nx-1)/2 )**2 +
        ( y - (ny-1)/2 )**2 +
        ( z - (nz-1)/2 )**2
      dist = dist**0.5
      density[(z*ny + y)*nx + x]
           = 1/(2**(dist+2))
```

Fig. 3: `Python` script to generate a smoke sphere. In the actual implementation, we restrict the sphere radius to $\frac{9}{2}$ and enforce the edges to be 0. This reduces harsh edges around the medium boundary.

script to output grid densities into a testing scene. The script pseudocode to generate a ball is presented in figure 3. Other than this, and cloud data included with `pbrt`, we use no external data.

V. Results

An example of table with fog is presented in figure 4. An example ball of fog is presented in figure 5, and a gaussian-randomized smoke cloud is presented in figure 6. Finally, we model the ajax bust with fire (similar to the yellow smoke in figure 6) in figure 7 and underwater in figure 8.

VI. Conclusion

Overall, this was a fun, yet challenging, project. We are grateful for the incredible pbr book and the provided `pbrt` source code. We were successful in implementation, and created some interesting renderings.

References

[1] J. T. Kajiya, "The rendering equation," in *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '86. New York, NY, USA: Association for Computing Machinery, 1986, p. 143–150. [Online]. Available: https://doi.org/10.1145/15922.15902

[2] E. P. Lafortune, "Mathematical models and monte carlo algorithms for physically based rendering," Ph.D. dissertation, Department of Computer Science, KU Leuven, Celestijnenlaan 200A, 3001 Heverlee, Belgium, February 1995.

[3] E. P. Lafortune and Y. D. Willems, "Rendering participating media with bidirectional path tracing," in *Rendering Techniques '96*, X. Pueyo and P. Schröder, Eds. Vienna: Springer Vienna, 1996, pp. 91–100.

[4] M. Pharr, W. Jakob, and G. Humphreys, *Physically Based Rendering: From Theory to Implementation*, 3rd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016.

Fig. 4: Table scene (Olesya Jakob) with fog. This example uses a homogeneous medium. $\sigma_a = [0.002, 0.002, 0.0001]$, $\sigma_s = [0.075, 0.025, 0.01]$, $g = 0.9$.

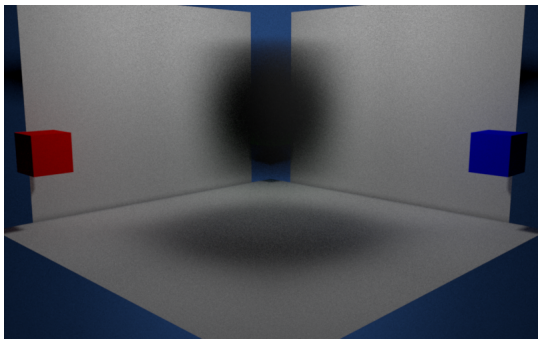


Fig. 5: Ball of smoke modeled using heterogeneous medium. $\sigma_a = [90, 90, 90]$, $\sigma_s = [10, 10, 10]$, and $g = 0$.

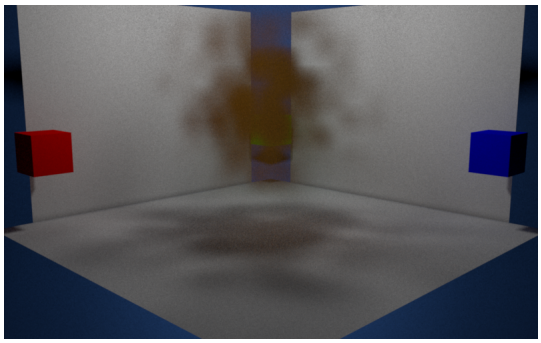


Fig. 6: Area of yellow smoke modeled using heterogeneous medium. $\sigma_a = [2.4, 4.5, 7.2]$, $\sigma_s = [5.6, 3.5, 0.8]$, and $g = -0.7$.

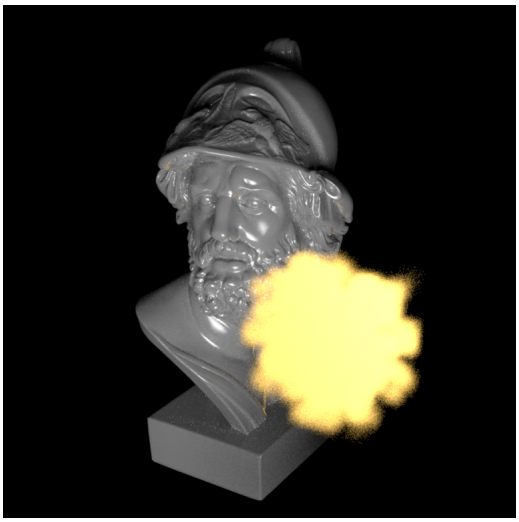


Fig. 7: Ajax bust modeled with a ball of exploding fire. Ajax uses a microfacet BSDF. Notice the orange specular highlights made by the fire.

Fig. 8: Ajax bust modeled to appear underwater. Ajax uses a dielectric BSDF with interior index of refraction $\eta = 1.33$ and exterior index of refraction $\eta = 1.5$. We use $\sigma_a = [0.50.250.01]$, $\sigma_s = [0.010.260.5]$, and $g = -0.1$ with 4096 samples per pixel. Note that a simple diffuse BSDF for Ajax with a blue appearance would achieve a similar, but less interesting, result.