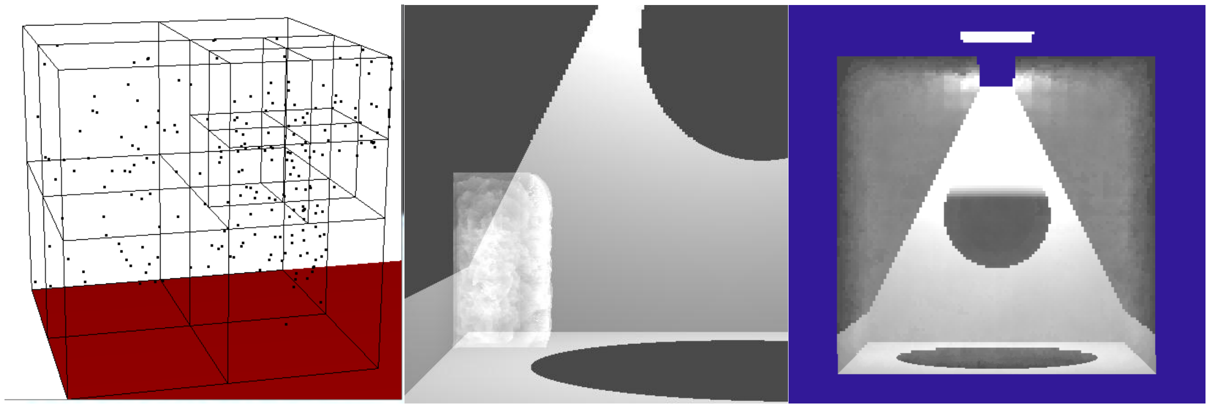
Simulating Smoke with an Octree Data Structure and Ray Marching

Edward Eisenberger Maria Montenegro



Abstract

We present a method for simulating and rendering smoke using an Octree data structure and Monte Carlo Ray tracing (ray marching).

1 Introduction

We investigated the technical problem of simulating and rendering compressible fluid, in particular smoke. Our project was a two-fold plan. We first implemented a three dimensional particle system to model the motion of smoke accurately. Second we rendered light's interaction with smoke, such as the translucency, shadows, and color.

Simulating natural phenomena, such as smoke, is a challenging problem in the computer graphics field because they contain complex and turbulent motions along with rich visual detail. This becomes increasingly difficult and expensive when interacting with objects and on high resolution grids.

2 Related Work

Our project cemented its foundation by merging two previous assignments from the Advanced Computer Graphics course at Rensselaer Polytechnic Institute, the fluid portion of "Assignment 2: Cloth & Fluid Simulation" and Ray Tracing with a similar concept of Photon Mapping from "Assignment 3: Ray Tracing, Radiosity, & Photon Mapping." We also incorporated an OCTree data structure for both simulating and rendering particles following the model described in "Simulating Water and Smoke with an Octree Data Structure"

2.1 Fluid Simulation

We kept the concept of representing smoke as particles in a grid. Simulating the motion of these particles is usually modeled in a uniform grid using the Navier-Stokes equations; however we chose to model the motion of particles using Octrees as described in "Simulating Water and Smoke with an Octree Data Structure."

2.2 Ray Tracing

We decided to expand our Hw 3 ray tracer by including a ray marcher algorithm. When a ray is cast through the media, we march down the ray using equal size steps collecting particles at each step to calculate the radiance and scattering at that point.

3 Technical Details

In this section we describe the data structures and algorithms implemented to make this project possible.

3.1 Data Structures

To lay the framework, we utilized several data structures from the Advanced Computer Graphics course assignments, such as Vec3f, Bounding Box, Edge, Face, Mesh, Ray, and Ray Tracer. Then we updated these as well as implemented several data structures to approach this problem.

3.1.1 Bounding Box

We treat bounding boxes as cells. Each one contains the u, v, and w face velocities as well as pressure. The bounding box is the building block for our entire scene. It creates the space in which our particles move around.

3.1.2 Octree

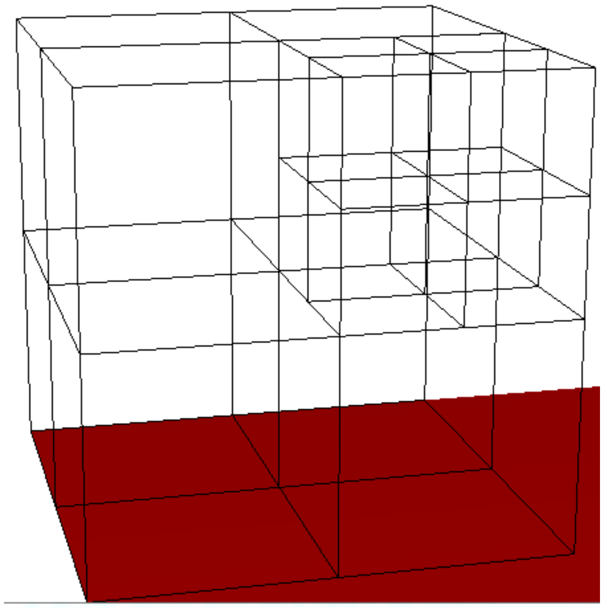


Figure 1 - Octree Visualization.

An Octree node is defined by a bounding box, which provides the overall size, pointers to each of its eight children, a split center, and depth. We use the depth to ensure we do not subdivide indefinitely. This data structure contains all the necessary functions to add and remove particles, merge children, split a node and create its eight children, and a cleanup method.

3.2 Algorithms

We implemented three main algorithms. The first generates the scene and places the particles. The second animates the particles according to Navier-Stokes equations. And the third renders the scene to show light interacting with smoke particles.

3.2.1 Generating the Scene and Particles

Our program reads in two files, a text file setting up the particle grid and an object file which creates the meshes (lights, floor, walls, etc.). The particle file declares a 3-space coordinate which is the max of the scene. A bounding box is then created from the origin to this max point and is the first node of our Octree and another boundingbox is created which is slightly larger, known as our “grid,” and is used to keep particles from leaving our grid. Next we initialize the particles with a few set commands from the Fluid Simulation assignment. Particles can start either everywhere within the grid or in the bottom left hand corner (0 <= y <= max_y, 0 <= x <= max_x). Particles are then generated according to these parameters specified and are added to the Octree which then subdivides accordingly. The Initial velocities for the grid can be set to either zero or random with the default set to zero. Lastly, any specific velocities can be set with a direction (u,v,w) at a particular position and a magnitude of the velocity. These velocities are then set to the cells in the Octree which contain the position specified.

3.2.2 Animating the Particles

The algorithm for animating particles follows the same general outline as the Fluid Simulation assignment. Our function, Animate(), is called in sets of 10 once animation has been initialized. It computes the new face velocities, sets the boundary velocities, updates pressures, copies over the new face velocities, moves the particles, reassigns the particles to the Octree, sets which cells in the Octree are empty, then draws the scene.

The function to compute new velocities contains the Navier-Stokes equations. We traverse through the Octree using depth first search and compute the new u face velocity. We then traverse the Octree twice more, once to compute the new v face velocity and once to compute the new w face velocity.

The function SetBoundaryVelocities() simples sets the grid face velocities to zero.

We update the pressure of each cell by using a Depth First search of the Octree and calculating the divergence at each node. The values are all discretized to account for the change in size of each cell. This is done by multiplying the values of cells next to the one we are currently looking at by the ratio of the size of our current cell over the size of the neighbor.

Once all of the values have been updated, we move each particle. This is done by another DFS through the Octree. At each node, we collect all of the particles then calculate the interpolated velocity at the particle's position and add it to the position times the timestep.

We get the interpolated velocity by looking at the cell which contains the point passed to us and the appropriate neighbors. We determine the neighbors in the same fashion as in the Fluid Simulation assignment. The difference comes into play with each of the values which need to be discretized to account for the changes in cell size.

Now that our particles have moved, we need to reassign them to the Octree. We use yet another DFS to traverse the tree. At each node we collect all of the particles and for each particle, we check if it is in the cell we are currently looking at. If it is then we move on, otherwise we remove it from that cell and add back to the Octree which then filters it down to the appropriate child cell. Once we have traversed the entire Octree, we call the cleanup routine.

Our cleanup routine is another DFS but instead of stopping at nodes, we stop at each node's parent. If the node we are looking at is a parent and the sum of its children's particles is less than a specified threshold, we merge the children. Otherwise we explore its children.

The merge routine is simple. We collect all of the particles, the face velocities, and pressures of each child node then add all of it to the parent and delete the children.

Once the tree has settled, we traverse the octree once more. If a bottom node has no particles, we set the cell's status to empty, otherwise full.

Now that the Octree has been taken care of, we pass the information to our rendering functions. These functions setup the appropriate VBOs to render the particles, octree, and scene.

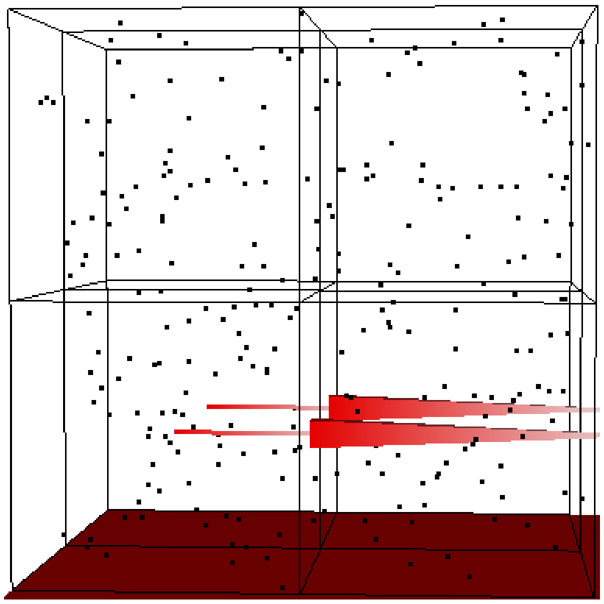


Figure 2 - Octree Visualization with Face velocities after initialization.

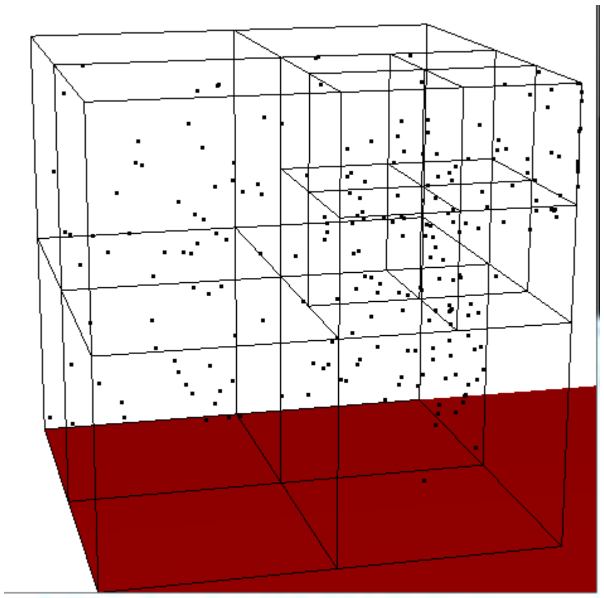


Figure 3 - Octree and Particle Visualization after several time steps.

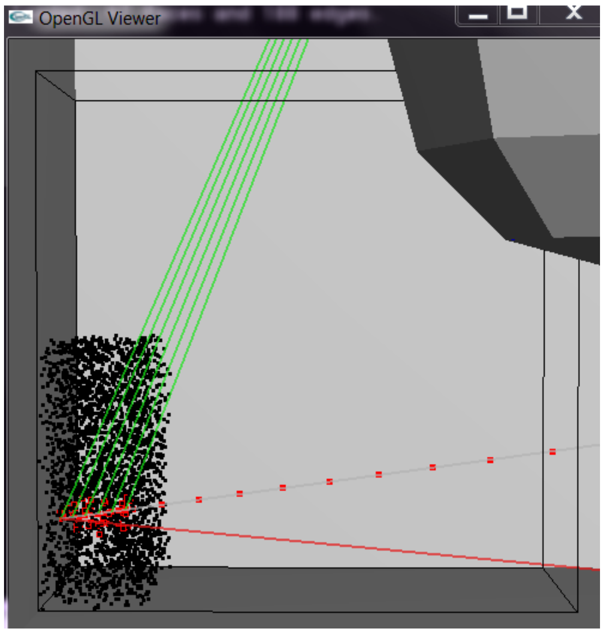
3.2.3 Rendering the Scene

Rendering participating media, in this case smoke, is complex thanks to the amount of detail that needs to be taking into account. For this paper we used the Monte Carlo ray tracing approach. Even though it suffers from noise and is very computationally intensive, it is straightforward to include surface scattering and anisotropic phase functions. The first step of our algorithm is ray marching. This is needed to be able to calculate the radiance in the media. The second part of the algorithm is efficiently calculating the radiance with single scattering and multiple scattering. In this paper we are only going to talk about single scattering.

3.2.1 Pre -Computation:

Before the scene is rendered, we go through all the Octree pre-calculating the radiance and transmittance of each bounding box with respect to the light source for future use in our inScattering calculation.

3.2.2 Ray Marching:



Ray marching consists of stepping through the ray calculating the radiance at each point. Before we start starting ray marching, we send a ray to the scene to see if we hit anything. If an object is hit, the first hit position and the ray's origin are stored to calculate the distance we are about to march. With the distance calculated, the step size can be computed. Usually the step size is calculating by dividing the total distance by the number of steps you want to do. This is not the only way to do it; you can have a random step size or a fit step size and random number of steps too. In this paper we are going to emphasize on equal step sized down the ray. After testing the three methods and getting similar results, we realized that equal steps size worked better for our particle gathering section.

We combined our ray marching implementation with a type of photon mapping, but instead of collecting photons we collect particles. We also set a required number of particles that have to be collected at each step to compute the radiance. When marching down the ray, we use a cylinder to represent the size of the volume we are marching down. The cylinder has a fixed width (our step size) and a random radius that will vary depending how many particles we need to collect. At each step, we gather the particles that are inside our cylinder with the use of our OCTree. If the number of particles that we need to gather is not matched, we increase the size of our radius and repeat until we have collected the necessary amount of particles inside of our cylinder. Once we have the right amount, we start calculating the radiance.

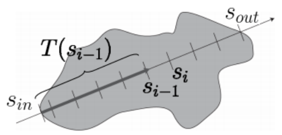
3.2.2 Calculating the Radiance and Single Scattering:

$$L(\mathbf{x},\vec{\omega}) = \int_0^s T_r(\mathbf{x}\leftrightarrow\mathbf{x}_t)\sigma_s(\mathbf{x}_t)L_i(\mathbf{x}_t,\vec{\omega})\,dt + T_r(\mathbf{x}\leftrightarrow\mathbf{x}_s)L(\mathbf{x}_s,\vec{\omega}),$$

To calculate the radiance at a point in the ray, we use the Radiative Transport Equation, where L is the radiance arriving at that point with direction w, s is depth of the media where we are (our step down the ray), T is the transmittance , sigma_s is the scattering coefficient and Li is the inScattering radiance at that point.

When solving for L(x,w), we first find the transmittance at point x (point where we are in the ray), which is basically a fraction of light transported a certain distance down the ray.

$$T(s_i) = T(s_{i-1})T(s_{i-1}\to s_i)$$
$$T(s_{i-1}\to s_i) = \int_{s_{i-1}}^{s_i} e^{-\sigma_t(s')}ds' \approx (1-\sigma_t)(s_i - s_{i-1})$$



Once we have T, we solve the inscattering radiance. The inscattering radiance is the sum of the single scattering and the multiple scattering. In this paper we only solve for the single scattering equation where p is the phase function, Lr is the reduced radiance, V is the visible function and H is the geometry term.

$$L_r(\mathbf{x}'\to\mathbf{x}) = T_r(\mathbf{x}'\leftrightarrow\mathbf{x})\,L(\mathbf{x}'\to\mathbf{x}).$$

$$L_s(\mathbf{x},\vec{\omega}) = \int_A p(\mathbf{x},\vec{\omega}',\vec{\omega})\,L_r(\mathbf{x}'\to\mathbf{x})\,V(\mathbf{x}'\to\mathbf{x})\,H(\mathbf{x}'\to\mathbf{x})\,d\mathbf{x}',$$

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

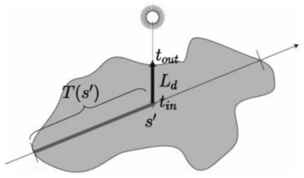
In this paper we assumed our media was isotropic, making our phase function a constant of p=1/π4.

Given that we are ray marching down the ray, we use the Monte Carlo equation to solve for the integration. For this we need to compute the probability of distribution function (pdf) and cycle through every face in the scene to be able to calculate our variables.

$$pdf(\mathbf{x}') = \frac{\vec{\mathbf{n}}(\mathbf{x}') \cdot \frac{\mathbf{x}_0 - \mathbf{x}'}{\|\mathbf{x}_0 - \mathbf{x}'\|}}{\|\mathbf{x}_0 - \mathbf{x}'\|^2},$$

$$L_s(\mathbf{x}, \vec{\omega}) \approx \frac{1}{N} \sum_{j=1}^{N} \frac{p L_r V H}{pdf(\mathbf{x}'_j)}.$$

After cycling for every face, we also compute internal scattering created by the light source inside the media. To calculate it we march from the point we are at towards the light, until we are out of the media. At each step, we calculate the transmittance at that point. Once we are out, we multiply the new transmittance by the light source and add this internal scattering to our Ls.



To accommodate for the particle gathering and radius increase, we cycle through all the particles collected, gathering there transmittance and radiance with the use of the pre-computed information. At this stage we apply a Gaussian blur filter. Then we divided by the density of the cylinder and add it to our final radiance.

Once we are done ray marching, we multiple our final L by the width of the step to account for the march and add it to the pixel color.

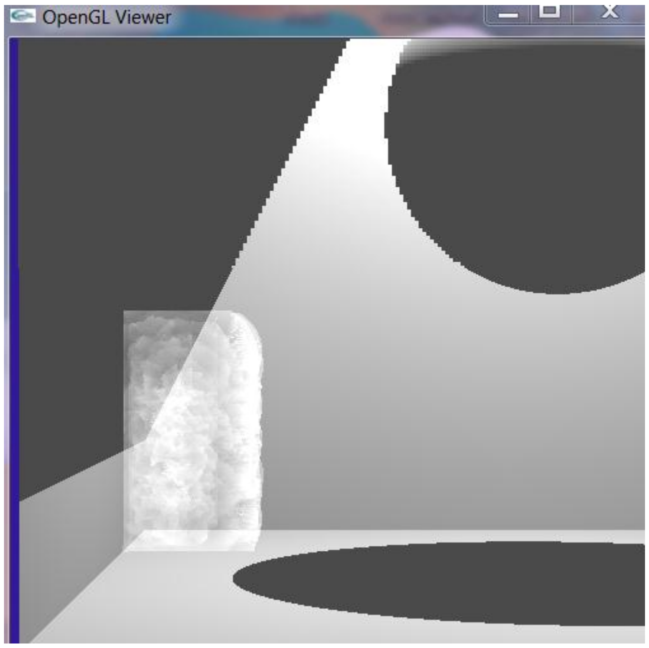


Figure 4:Smoke with Single Scattering

3.2 Core Features

The core features of our project are the accurate simulation of compressible particles and the rendering of smoke particles. Our main contribution to the simulation, and most difficult aspect, was the Octree implementation.

3.3 Challenges

We faced many challenges along the way. Implementing the Octree with all of the proper methods needed for an accurate ray tracer as well rendering smoke in general proved to be quite challenging.

3.3.1 Able to Overcome

One major challenge was implementing an Octree that constantly adapts. Creating the initial Octree was simple, requiring only the abilities to both add a particle and split a cell. Once particles began moving it became apparent that we needed to consider merging children which required the ability to remove particles. Merging children was a challenge and led to many memory leaks since we deal with pointers. The most difficult portion was solving the Navier-Stokes equations with varying sizes of cells. Discretizing each cell by multiplying the neighbor the size of the original cell divided by the size of the neighboring cell solved the issues with getting the interpolated velocities as well as computing the new Velocity.

3.3.2 Failed to Overcome

The Octree failed in two aspects, calculating pressure and removing sinks/sources. Discretizing worked for both computing the new velocity as well as the interpolated velocity but the pressure still has a tendency to explode after several steps. Particles also have a tendency to sink to one of the corners of the box after many time steps.

4. Results

Given the challenges and setbacks encountered, overall we were satisfied with the results. Figures 5 through 8 show a progression of our renderings.

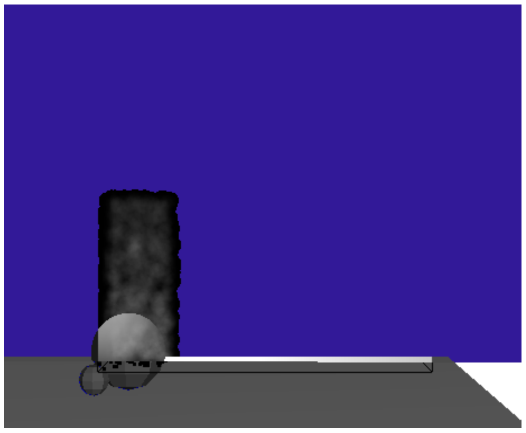


Figure 5: First Smoke Renderings (1)

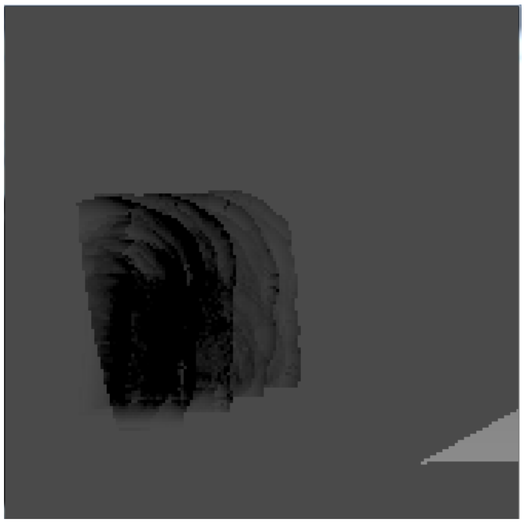


Figure 6: Smoke with Incorrect Scattering

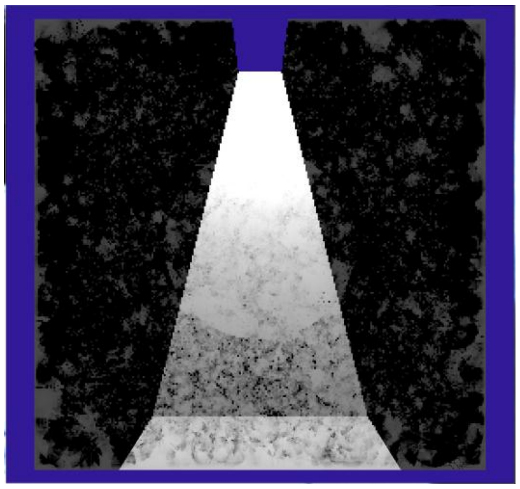


Figure 7 - Smoke in Cornell Box, incomplete scattering

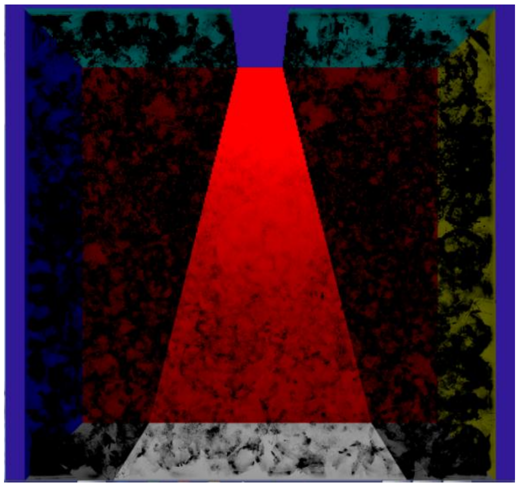


Figure 8: smoke with no scattering

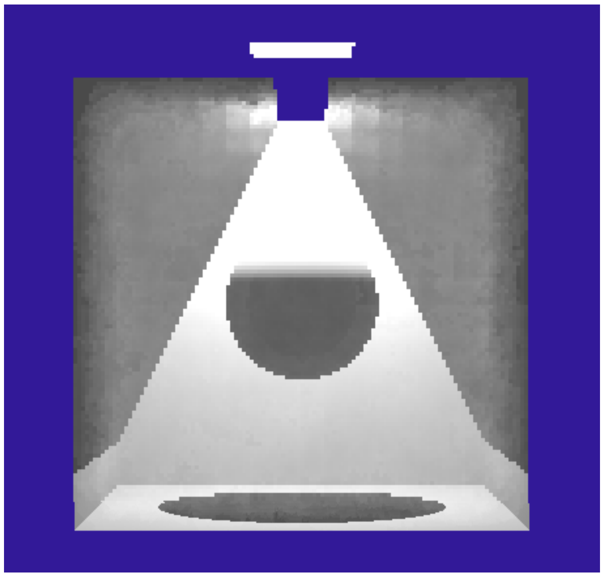


Figure 9: Smoke with Scattering in room full of smoke

4.1 Who did what?

Ed implemented the octree data structure and the simulation of smoke particles and assisted with debugging for the rendering.

Maria implemented the Monte Carlo Ray Tracing, ray marching, and rendering the smoke.

5. Conclusions

In this paper, we modeled small scale smoke scenes. However, we only implemented single scattering and would like to extend this application to include multi-scattering and fix the lingering bugs with the Navier-Stokes equations.

6. Bibliography

Fedkiw, Ronald, Jos Stam, and Henrik W. Jensen. *Visual Simulation of Smoke*. Tech. SIGGRAPH '01 Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques. Web. 29 Mar. 2012. <http://dl.acm.org/citation.cfm?id=383260>.

Losasso, Frank, Frederic Gibou, and Ron Fedkiw. *Simulating Water and Smoke with an Octree Data Structure*. Tech. SIGGRAPH '04 ACM SIGGRAPH 2004 Papers. Web. 29 Mar. 2012. <http://dl.acm.org/citation.cfm?id=1015745>.

Lubich, Chistrian, and Alexander Ostermann. *Runge-Kutta Time Discretization of Reaction-diffusion and Navier-Stokes Equations" Nonsmooth-data Error Estimates and Applications to Long-time Behaviour*. Tech. Applied Numerical Mathematics 22. Web. 29 Mar. 2012. <http://www.sciencedirect.com/science?_ob=MiamiImageURL&_cid=271992&_user=659639&_pii=S0168927496000384&_check=y&_origin=article&_zone=toolbar&_coverDate=30-Nov-1996&view=c&originContentFamily=serial&wchp=dGLbVlS-zSkWA&md5=7dfce92c2656f4ced963a197ac999ab0/1-s2.0-S0168927496000384-main.pdf>.

Rasmussen, Nick, Duc Q. Nguyen, Willi Geiger, and Ronald Fedkiw. *Smoke Simulation for Large Scale Phenomena*. Tech. SIGGRAPH '03 ACM SIGGRAPH 2003 Papers, July 2003. Web. 29 Mar. 2012. <http://dl.acm.org/citation.cfm?id=882262.882335>.

Shi, Lin, and Yizhou Yu. *VISUAL SMOKE SIMULATION WITH ADAPTIVE OCTREE REFINEMENT*. Tech. SIGGRAPH 01 Conference Proceedings. Web. 29 Mar. 2012.

<http://www.ann.jussieu.fr/~frey/papers/applications/Shi%20L.,%20Visual%20smoke%20simulation%20with%20adaptive%20octree%20refinement.pdf>.

Jarosz, Wojciech, Craig Donner, Matthias Zwicker, and Henrik W. Jensen. "Radiance Caching for Participating Media." *In ACM Transactions on Graphics (Presented at ACM SIGGRAPH 2008)*. ACM Transactions on Graphics (Presented at ACM SIGGRAPH 2008). Web. 3 May 2012. <http://zurich.disneyresearch.com/~wjarosz/publications/jarosz08radiance.html>.

7. APPENDIX A: Code

For complete code, please visit our public repository on github at:

https://bergermeister@github.com/bergermeister/Smoke-Simulation.git