Ray Tracing in Participating Media

Matthew Bonnecaze, Samuel Stuart

CSCI 4350 - Advanced Computer Graphics,

Rensselaer Polytechnic Institute

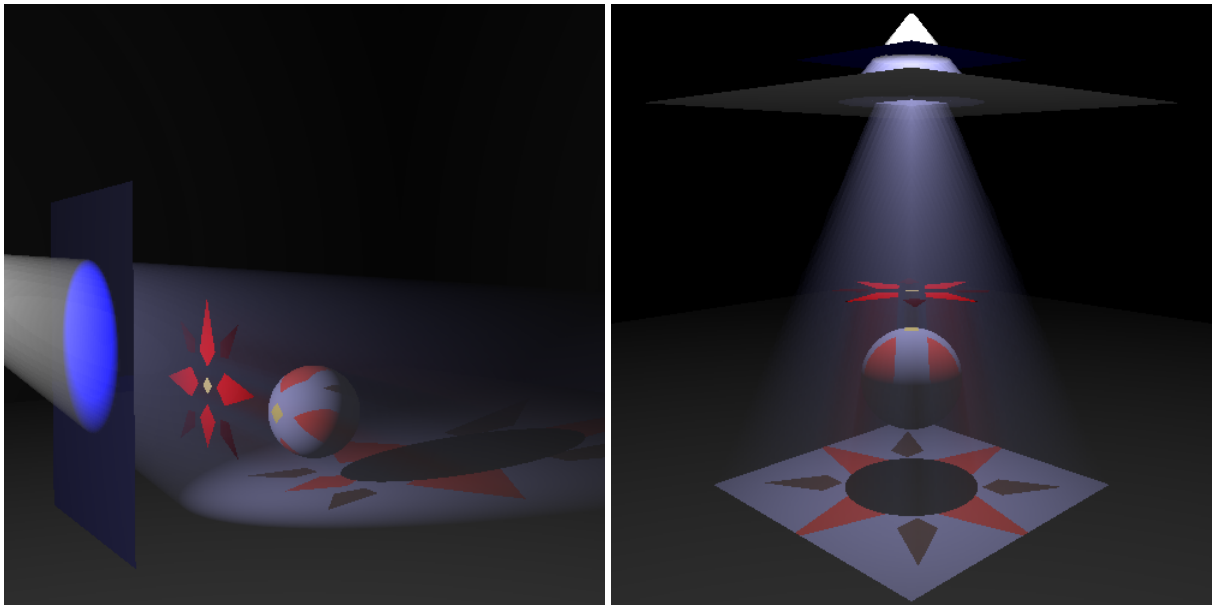

Figure 1: *Ray tracing participating media through stained glass. Note the colored light, color bleeding, and volumetric shadows.*

Abstract

We extend the CSCI 4350 HW3 code base to allow for ray tracing in participating media. To do so, we incorporate real-time shadows in OpenGL and accumulate radiance by ray marching on HW3's anti-aliasing rays. We also implemented raycasted shadows as a backup for shadow map shortcomings. Additionally, we extend HW3's light code to allow for spotlights and colored lighting through stained glass. This only approximates the effect of participating media by assuming light only scatters once, and a better approximation can be achieved by incorporating multi-scatter algorithms.

1 Introduction

Beams of light are often visible in daily life: car headlights in the rain, a window on a sunny day in a dusty room, a flashlight illuminating the smoke of a campfire, see figure 2. However, standard ray tracing techniques can only calculate the effect of light on the surface of objects, not in the volume between them. Much research has been done on replicating this effect, called participating media, in computer graphics, and as a result, there are many techniques for approximating it.

2 Prior Work

A real-time technique was introduced in [Tóth and Umenhoffer 2009] which used ray-marching to approximate the effect of participating media. To achieve real-time speeds, they used shadow mapping for visibility calculations (introduced in [Williams 1978]) and took interleaved samples, as described in [Keller and Heidrich 2001]. Their calcuations are described in section 3.1. Since we are not aiming for real-time speeds, we did not use interleaved sampling. However, we implemented shadow mapping in OpenGL, using [Williams 1978] as a guide.

This technique first renders a image from each light source's perspective which encodes the depth of objects to the light as darkened pixels in a texture. Another render pass from the camera's perspective compares the z-position on visible surfaces to the depth texture (after some matrix projection), and renders shadows if this z-value is less than the texture value.

[Wyman and Ransey 2008] developed an approach to render shadow volumes in homogeneous single scattering media that also utilizes shadow maps and ray marching but improves the efficiency of rendering shadow volumes by only performing ray marching in areas in the shadow map that have shadows. [Wyman and Ransey 2008] were also able to modify this method to apply textures to the spotlights, which adds multiple colored light volumes to the render.

Thus far we have only looked at single scattering effects in participating media using ray marching. [Premože et al. 2004] proposed an approach to approximating multiple scatter ray marching by voxelizing light volumes and computing the attenuated radiance for each light source in a precomputation step. This

Figure 2: *Image by Bonnecaze, M.: the Pantheon in Rome, a clear example of participating media*

approach in [Premože et al. 2004] also allows for rendering effects in non-homogeneous participating media.

3 Mathematical Basis

3.1 The Radiative Transport Equation

The mathematical core of the participating media effect is the radiative transport equation. A ray of light can be represented by the equation $\vec{x}(s) = \vec{x}_0 + \vec{\omega}s$, where $\vec{x}_0$ is the starting position, $\vec{\omega}$ is the direction, and s is a parameter to represent the distance along the ray. In homogeneous participating media, which does not emit light, the radiative transport equation gives the change in radiance, L, along the ray as:

$$\frac{dL(\vec{x}(s),\vec{\omega})}{ds} = -\tau L(\vec{x}(s),\vec{\omega}) + \tau a \int_{\Omega'} L(\vec{x}(s),\vec{\omega}')P(\vec{\omega}',\vec{\omega})d\vec{\omega}'$$

where τ is the density of the participating media (the chance of collision), a is the albedo (the chance of scattering upon collision), $\vec{\omega}'$ is the direction to the light source, and $P(\vec{\omega}',\vec{\omega})$ is a phase function which produces the probability density of the direction of light scattering.

Since this equation is too difficult to solve in real-time, we simplify the equation by assuming that light can only scatter once. Following the work in [Tóth and Umenhoffer 2009], this assumption produces the approximation $L_i(\vec{x}(s),\vec{\omega}) \approx \tau a \int_{\Omega'} L(\vec{x}(s),\vec{\omega}')P(\vec{\omega}',\vec{\omega})d\vec{\omega}'$, where L_i is the accumulated radiance from one scatter. Using the fundamental theorem of calculus and solving analytically, as in [Tóth and Umenhoffer 2009], produces the approximation we use:

$$L_i(\vec{x}(s),\vec{\omega}) = \tau a \frac{\Phi}{4\pi d^2} v(\vec{x}) e^{-\tau d} P(\vec{\omega}_l,\vec{\omega}) \tag{1}$$

where $v(\vec{x})$ is a visibility function that returns 1 if x is fully illuminated and 0 if x is fully in shadow, d is the distance to the light source, Φ is the power (energy) of the light source, and both e and π have their standard mathematical values.

3.2 Spotlights

A spotlight is a light source that emits light in a specific direction in a defined conical shape. The angle of the cone (θ_{cutoff}), known as the cut-off angle or the sector's angle, can range between 0 degrees and 90 degrees. To implement spotlighting, we needed to determine if a point (p) is within the spotlight's cone.

As each light source is represented by a quad in the scene, we use the normal of the face to determine the spotlight's direction. We will call the direction of the spotlight d.

We then need to determine the direction of point p to the centroid of the spotlight, c. Let this value be v. This can be calculated as follows:

$$v = c - p \tag{2}$$

Using this, we can determine the cosine of the angle between d and v.

$$\cos(\theta_{dv}) = d \cdot v \tag{3}$$

Finally, to check if point p is within the cone of the spotlight, the following must be true:

$$\cos(\theta_{dv}) > \cos(\theta_{cutoff}) \tag{4}$$

If this is true, we can calculate how far inside the cone point p is to get the value i, the intensity of the spotlight at point p.

$$i = 1 - \frac{1 - \cos(\theta_{dv})}{1 - \cos(\theta_{cutoff})} \tag{5}$$

where i is a floating-point value between 0 and 1.

3.3 Colored Lights

We wanted to implement a stained glass effect by coloring illuminated geometry and participating media based on semi-transparent materials between this point and the light source. To do this, we added an alpha (opacity) value to the Materials class, with an alpha of 1 being fully opaque and an alpha of 0 being fully transparent. To add two colors together, we decided to use the following equation, as it factors in the alpha of the Material to prevent the resulting sum of the two color from being greater than [1,1,1], which would be an invalid color value.

$$b_1 = b_0(1 - \alpha) + c\alpha \tag{6}$$

where b_1 is the resulting accumulated color ("blend") value, b_0 is the current blend value, c is the color being blended into b_0, and α is the opacity value (between 0 and 1) of c.

4 Implementation

As mentioned, we take advantage of the processes and data structures present within HW3. Particularly, we make heavy use of the Ray, Hit, Raytracer, and OpenGLRenderer classes.

4.1 Spotlights

In the provided code, all light sources are assumed to be point light sources, meaning that the scene is illuminated from all directions. Spotlights are a common depiction of volumetric lighting, as spotlights have a defined conical shape that is emitted in a defined direction. Light sources are defined as a face with a material with an emission value greater than [0,0,0]. To specify the cut-off angle in the obj files, a new token "c" was added. The cut-off angle must be a floating-point value between 0 and 90. If this token is included, the light source is stored as a spotlight source.

We use equations (2), (3), and (4) to determine if a point is within the spotlight. To determine the intensity of the light, assuming that the point is within the spotlight's cone, equation (5) is used to produce an intensity value between 0 and 1. If a spotlight is being used, this intensity value is calculated and multiplied to areas where the point light source was used to clamp the illumination to be within the spotlight's cone.

4.2 Ray Marching

As mentioned in [Tóth and Umenhoffer 2009], simulating participating media by ray marching can be done as a post-processing effect. Consequently, we run a ray marching function on each anti-aliasing ray produced by the HW3 code base. This function takes the ray, current pixel color, and hit data (storing the ray parameter s of the nearest hit) as input, and produces an adjusted pixel color that accounts for participating media effects.

To do so, we calculate the light accumulation at discrete steps along the ray, using equation (1). The length of these steps is determined by a command-line parameter, `ray_march_step`. The density τ and albedo a are determined by command-line parameters `air_density` and `air_albedo`. We make sure to accumulate light at the start of the ray, and we make the further simplifying assumption that light scatters equally in every direction, producing a constant phase function.

Thus, the function requires access to the light position and intensity (accessible through `GLOBAL_args->mesh->GetLights()`), as well as a method to calculate the visibility of any given position along any given ray. This visibility functionality is provided by shadow mapping or shadow rays, as described in sections 4.3 and 4.4. The results of ray marching while using only spotlight intensity as a visibility function are provided in figure 3. We found visually interesting results with physically impossible albedo values, such as 10.0.

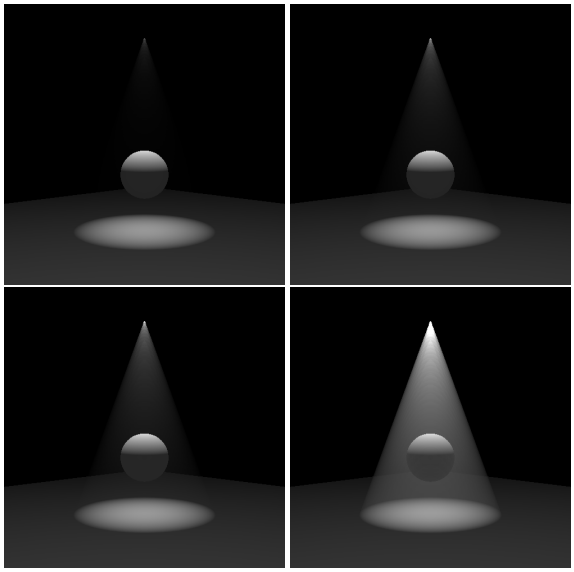

Figure 3: *Ray marching using spotlight intensity as visibility. Each image has a ray march step value of 0.1 and an air density value of 0.1. From left to right, top to bottom, the albedo values are 0.1, 0.5, 1.0, and 10.0.*

4.3 Shadow Mapping

To extend HW3 to use shadow maps, extensive use of OpenGL is required. We make two changes to the processing pipeline in `OpenGLRenderer.cpp`: a new setup stage and a new render stage. In the setup stage, a framebuffer is configured to read depth information, and a texture is configured to store depth information received from the frame buffer. The data type of the texture must be `GL_DEPTH_COMPONENT`. After initializing the framebuffer to an unsigned int `depthMapFBO` and the texture to an unsigned int `depthMap`, the following code binds the two together for later use in the rendering pass:

```
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glFramebufferTexture2D(GL_FRAMEBUFFER,
                       GL_DEPTH_ATTACHMENT,
                       GL_TEXTURE_2D, depthMap, 0);
```

The framebuffer must then be unbound as the primary framebuffer before proceeding.

The other sections of the setup stage, such as configuring vertex buffer objects, proceed as normal. Then, before the scene is drawn, the new render stage takes place. Similar to a standard rendering pass, an MVP - Model, View, Projection - matrix is constructed. However, this MVP matrix translates a position in world space into a position in the light source's view space. As such, this matrix is typically referred to as the `depthMVP` matrix. In our implementation, the model matrix is the identity. Since we are using spotlights, the projection matrix is a perspective distortion matrix constructed by `glm::perspective<float>(45.0f, 1, 2, 50.0f)`. Lastly, the view matrix is constructed using the `glm::lookAt` function, creating a view matrix from the light's position facing the direction of its normal. However, if the light's normal vector is parallel to the camera's up direction, this function fails. We detect this case by checking if the cross product of these directions is zero, and in that case, we rotate the camera's up direction by 90 degrees about the z-axis before using `glm::lookAt`.

With the `depthMVP` matrix constructed, we can render the scene from the light's perspective. After rebinding the framebuffer and adjusting the viewport to match the colors and dimensions of the depth texture, we use HW3's `drawVBO(mvp, m, v)` function with the `depthMVP`, identity matrix, and depth view matrix as parameters. This populates the framebuffer with depth information from the light's perspective (due to the `depthMVP` transform) and writes it to the texture (from the earlier pseudocode.) A sample depth texture is provided in figure 4. Then, we unbind the framebuffer, revert the viewport, and clear the depth and color buffers. Finally, we render the scene from the normal perspective.

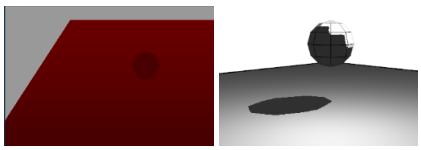

Figure 4: *The depth map (left) and resulting shadows (right) of a sample scene. The depth map is contrast-adjusted to be more visible.*

In order to see the shadows in real-time, we must send `depthMVP` to the vertex and fragment shaders. This is only done to visualize the results of only the shadow map, and is not necessary for participating media effects in the ray tracer. Using the x and y values from the vertex position transformed by `depthMVP`, we read the texture and compare its value to the transformed z-value. If the fragment has a lower z-value than the texture, it is below the depth the light sees at that position, and its color is set to black. A small epsilon bias is subtracted from the z-value to prevent shadow acne. The fragment and vertex shaders produce the results seen in figure 5. Lastly, we write a function to retrieve and sample the depth texture value and compare to the transformed z-value for any

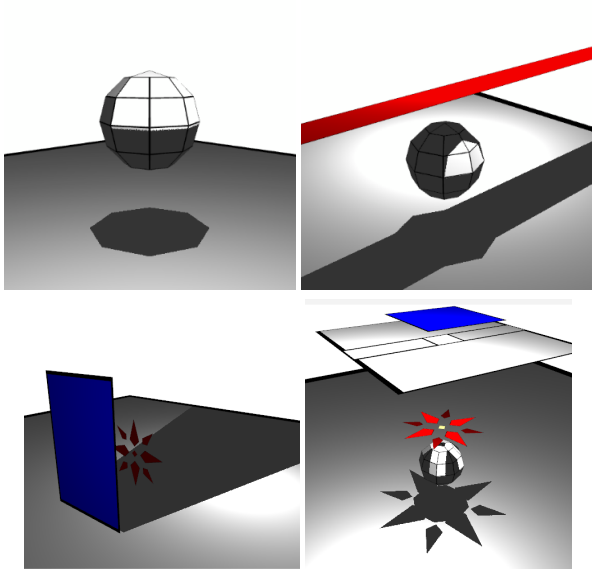

Figure 5: *Shadow Mapping in OpenGL. This is done as a precomputation before ray tracing and runs in real-time on an Intel(R) UDH Graphics 630 card.*

given position in world space. Unfortunately, our implementation of this function is incorrect. In the fragment shader code, the GLSL function `texture` can be used for texture lookup, but when ray tracing, we had to reconstruct the operation of this function in C++. Our best attempt produced the correct shadows, but at a position far too close to the light. This effect can be seen in figure 6. As a workaround to produce the desired effect, we cast shadow rays for each step in the ray march, but this approach is very slow and scales poorly. See section 5.1 for more details on how this workaround affects the scalability and render time of our program.

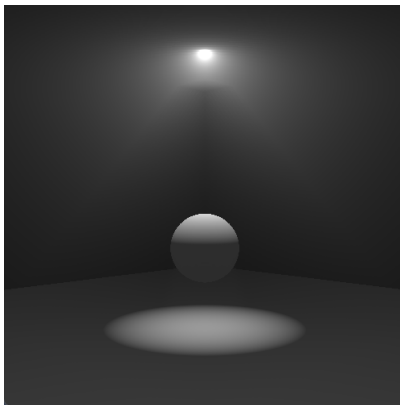

Figure 6: *Our closest failure using shadow maps in the ray tracer. The position of the sphere is incorrectly translated upward in light space, but the matrix transform involved means that applying a counteractive translation is not straightforward. Spotlight visibility checking was disabled for this render, which has a ray march step size of 0.1, an air density of 0.1, and an air albedo of 0.9.*

4.4 Ray-Cast Shadows

In order to improve the realism of our volumetric lighting, we utilized ray tracing to produce shadows and ray casting to produce volumetric shadows.

4.4.1 Ray Casting

In ray casting, a ray is projected from a point in a given direction to check for intersections, recording data about what the ray intersected with in a Hit data structure. The implementation of ray casting that we use checks for intersections between the ray and quads and primitives (i.e. spheres). This requires every quad and primitive to be iterated over during a single ray cast.

4.4.2 Ray Traced Shadows

We use ray tracing to determine the color of a pixel. If the initial ray from the camera intersects with a quad or a primitive, we need to determine if the object at that point is in shadow. This is done by casting a ray to each light source to determine the visibility of the point. Because of the cost of using multiple light sources and our initial plan to use shadow mapping with our ray marching implementation, we only support the use of a single light source. If no intersection occurs during this ray cast, then that point is not in shadow and the contribution from the light is applied to the pixel's color. Conversely, if an intersection occurs, then the light contribution is not applied. This allows for shadows to be applied onto the geometry in the scene.

4.4.3 Volumetric Shadows

While ray tracing allows for the addition of shadows, this will not produce volumetric shadows in the participating media. To implement this effect, as demonstrated in figure 7, for each step during ray marching, a ray is cast from the current step position to the light source. The contribution of that step will either be multiplied by [1,1,1] (white) if there were no intersections and by [0,0,0] (black) if the current step is in shadow. The color is accumulated through ray marching, so this addition allows for the appearance of shadows in the participating media. For efficiency, we only only perform this ray cast if the current step is within the spotlight's cone.

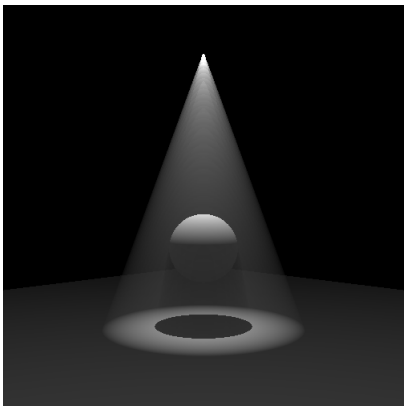

Figure 7: *A spotlight over a sphere. The sphere produces a hard shadow while a visible volumetric shadow can be seen beneath it. The hard shadow is produced by ray tracing while the volumetric shadow is produced as a result of ray marching and ray casting at each step to determine visibility. To produce this image, the distance between each ray march step is 0.1, the air density is also 0.1, and the air albedo is 5.*

Our naive implementation of volumetric shadows was inspired by the method introduced in [Wyman and Ransey 2008]. They used a shadow map in tandem with ray marching to produce shadow volumes, where a shadow map is rendered to determine areas of occlusion. This area is then used to determine if a step in the ray march is in shadow. As we were unable to finish our implementation of a shadow map, we opted to use ray casting to determine the visibility

at each step on the ray march. This is significantly more expensive than using the shadow map, since each ray cast requires us to iterate over every quad and primitive in the scene. However, one benefit of our approach is that we can render models that are not airtight, unlike the method proposed by [Wyman and Ransey 2008].

4.5 Colored Lights

We extend our implementation of light volumes and shadow volumes with the ability to include semi-transparent materials into a scene, which adjusts the color of participating media to give a stained-glass-like effect.

To do this, we added a new attribute to the Material data structure called "alpha", which is a floating-point value between 0 and 1, indicating how transparent or opaque the material is. An alpha value of 0 indicates that the material should be treated as completely transparent while an alpha value of 1 indicates that the material should be treated as completely opaque. We will refer to a quad or primitive with an alpha value less than 1 as a "glass object." By default, a material will have an alpha value of 1. To define a material's alpha value, the "alpha" token was added and is included in the material's definition in the obj file.

4.5.1 Modifications to Hit Data Structure

We modified the Hit data structure to include two new attributes: "blend" and "blended."

The blend value is the total accumulated color by intersecting quads and primitives. By default, the blend is [1,1,1] (white). The blend can either be directly set to a specific color value or be computed by adding a color to the total blend value. When adding a color to the blend value, we use the intersected geometry's diffuse color and alpha values. To add this color to our blend value, we use equation (6). The blended value indicates whether or not the blend value was set to a specific color or is an accumulation of colors.

4.5.2 Modifications to Ray Casting

During ray casting, if a glass object is intersected, we add its material's diffuse color to the passed in Hit's blend value. Intersecting with glass objects will not be counted as an intersection; intersections are only counted for quads or primitives with an alpha value of 1. In that case, the Hit's blend value is instead set to black [0,0,0], indicating that the point is in shadow.

4.5.3 Modifications to Ray Tracing

During the check to determine if a point is occluded in the ray tracing algorithm, the ray cast accumulates the contributions of the glass objects on the color of the point through the blend value. If the ray cast does not report an intersection with a non-glass object, the ray tracing algorithm applies the contribution of the light at this point. To include the accumulated color as a result of passing through glass objects, we multiply the resulting blend value to this shade value. This produces colored illuminated areas that are colored based on the accumulated colors of the glass objects, see figure 8.

4.5.4 Colored Light Volumes

Lastly, we multiply the contribution of the ray march step by the blend value based on the visibility reported by the ray cast. Recall the blend value can span between [0,0,0] and [1,1,1] and is set to [0,0,0] in the event of an occlusion. Therefore, this modification allows for both the shadow volumes in participating media as well as the ability for glass objects to color the participating media, see figure 9. Like the previous shadow volume implementation, we only only perform this ray cast if the current step is in the spotlight's cone to reduce the number of ray casts we need to do during a ray march.

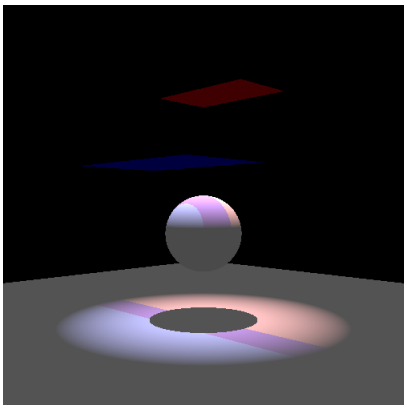

Figure 8: *A spotlight above two glass objects, one red and the other blue, coloring the illuminated areas. The colors of the illuminated areas correspond to the colors of the glass objects between these areas and the light source. The areas underneath both the red and blue glass objects result in a blend of the two colors, coloring the area purple.*

Building off of the shadow map approach to volumetric shadows proposed in [Wyman and Ransey 2008], we could similarly adapt a shadow map implementation to consider the color at each point in the map contributed by glass objects, in addition to recording occlusion caused by geometry with an alpha of 1.

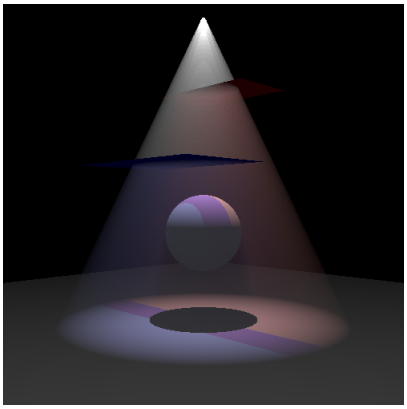

Figure 9: *A light volume colored by red and blue glass objects. Similar to the blending shown in figure 8, the participating media underneath the red and blue glass objects are colored purple. For this render, the distance between each ray march step is 0.1, the air density is 0.1, and the air albedo is 5.0.*

5 Shortcomings and Limitations

Our approach does produce impressive results, but we encountered issues in implementing shadow maps and multi-scatter effects.

5.1 Shadow Mapping and Multiple Light Sources

As mentioned in section 4.3, our texture lookup function for shadow maps is incomplete. This is a major bottleneck in efficiency improvements to our program, because raycasted shadows scale with order $O(s \cdot a \cdot r \cdot N \cdot M)$, where s is the amount of light sources in a scene, a is the amount of anti-alias rays, r is the amount of ray-march steps per ray, N is the pixel height, and M is the pixel width. However, shadow maps only scale with $O(s \cdot N \cdot M)$ and

can therefore render more complex scenes far more quickly. There does exist a difficulty with extending shadow maps to multiple light sources, since each light source would require a new render pass and a new texture. Our approach is currently only set up to store one depth texture, but could be feasibly extended to use an array of depth textures.

5.2 Multi-Scatter

Our implementation of ray marching assumes that the participating media is single scatter and homogeneous. However, this assumption is incorrect for effects such as fog and smoke. Once we had single scatter ray marching implemented, we decided that we would like to implement multiple scatter ray marching in order to to render these non-homogeneous and multi-scatter cases. However, naively, multi-scatter ray marching would be too computationally expensive to do. [Premože et al. 2004], however, proposed an approach to produce multiple scattering effects in participating media by introducing a precomputation step to speed up the rendering process. During this precomputation step, light volumes are voxelized and the attenuated radiance for each light source is computed, creating a lookup table for the available light for redistribution. At this point in the rendering step of the multi-scatter ray marching algorithm proposed by [Premože et al. 2004], the contribution by multi-scattering is approximated by summing the precomputed values along the path.

Unfortunately, the precomputation step proved to be challenging to implement, especially when trying to understand and recreate the data structures used in the paper. We attempted to extrapolate this information and produce multiple scattering effects in participating media without the precomputation step. However, the resulting implementation produced renders identical to our implementation of single scatter ray marching while taking significantly longer to produce the render.

6 Results

Even without shadow mapping and multi-scattering, we were able to produce visually impressive results using the functional portions of our program that highlight multi-colored light in participating media. Figures 10 and 11 (also in figure 1) took roughly 15-20 minutes to render.

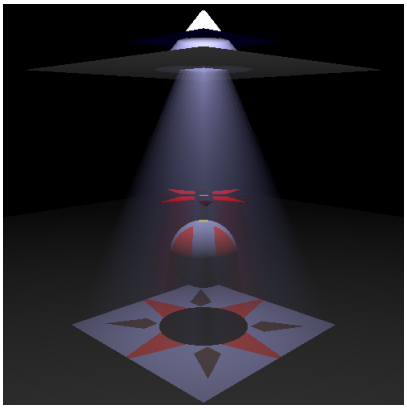

Figure 10: *The final result of our program. Note how the light reflecting off of the participating media changes color when passing through glass objects, and how the shadows are colored by the glass objects. The glass is not rendered as transparent to highlight the participating media effect. For this render, the distance between each ray march step is 0.1, the air density is 0.1, and the air albedo is 5.0.*

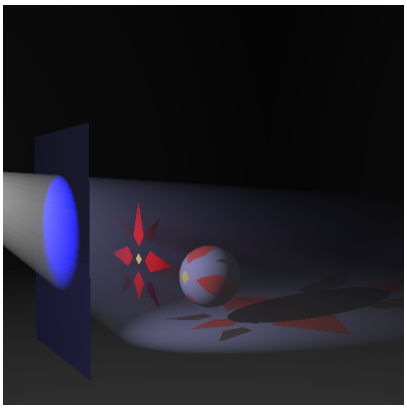

Figure 11: *Note that our approach works for any light angle. To produce this render, the distance between each ray march step is 0.1, the air density is 0.1, and the air albedo is 10.0.*

7 Conclusion and Future Work

Overall, our participating media effect is convincing for colored light and shadows, even when using unrealistic albedo values. However, these renders take much longer than standard ray tracing. We suspect that render time would be reduced substantially if shadow maps were functional, since they do not scale with increased ray marched steps. For future work, fixing the shadow map lookup function is paramount, and afterward, care should be taken to implement an array of depth textures to account for multiple light sources. Additionally, a correct implementation of multi-scatter effects would certainly slow down the program, but it would be a valuable comparison for the accuracy of participating media effects.

8 Work Breakdown

Matthew Bonnecaze created test models, implemented both spotlights and colored lighting, and attempted to implement multi-scatter ray marching. Samuel Stuart implemented ray marching and shadow mapping. Both contributed substantially to the paper and presentation. This project took approximately 80 hours of work, and was split fairly evenly between both group members.

References

KELLER, A., AND HEIDRICH, W. 2001. Interleaved sampling. in rendering techniques. *Twelfth Eurographics Workshop on Rendering*, 269–276.

LAFORTUNE, E., AND WILLEMS, Y. D. 1996. Rendering participating media with bidirectional path tracing. *Seventh Eurographics Workshop on Rendering*, 91–100.

PREMOŽE, S., ASHIKHMIN, M., RAMAMOORTHI, R., AND NAYAR, S. 2004. Practical rendering of multiple scattering effects in participating media. *Eurographics Symposium on Rendering*.

TÓTH, B., AND UMENHOFFER, T. 2009. Real-time volumetric lighting in participating media. *Eurographics 2009 - Short Papers*.

WILLIAMS, L. 1978. Casting curved shadows on curved surfaces. *ACM SIGGRAPH Computer Graphics*, 270–274.

WYMAN, C., AND RANSEY, S. 2008. Interactive volumetric shadows in participating media with single-scattering. *IEEE Symposium on Interactive Ray Tracing*, 87–92.

9 Fun Bugs

Most bugs in this project resulted in completely black or completely white images, but occasionally, the bugs were fun.

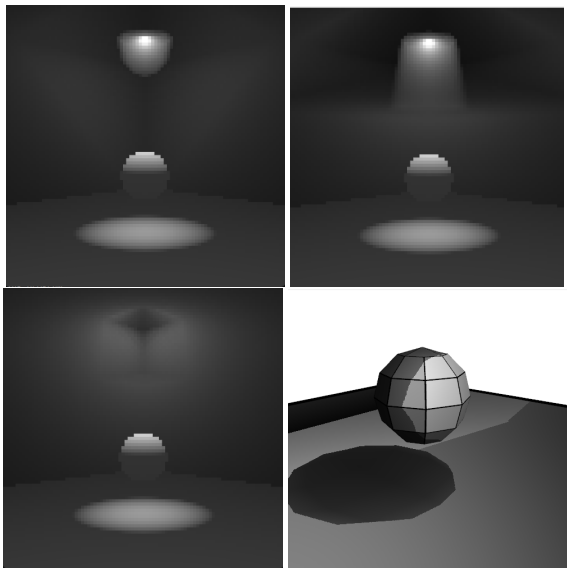

Figure 12: *Bugs with shadow mapping due to incorrect texture lookup and matrix transformations.*

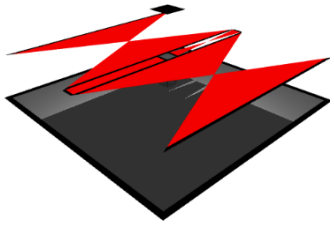

Figure 13: *Bug resulting from a failure to convert the vertex order from the exported obj file from Blender project to the format required for HW3.*