Bidirectional Monte Carlo Ray Tracing

Matthew Wong
University of Maryland, College Park
CMSC740

May 2020

1 Introduction

For my final project, I implemented Bidirectional Ray Tracing with Multiple Importance Sampling. I used Veach's article, Professor Zwicker's Introduction to MC Rendering article, and the PBRT Book to implement this algorithm. Some advantages of bidirectional path tracing to unidirectional path tracing is faster convergence and a better approach to handling scenes with small light sources and light sources that are not visible to the eye. The BDPT algorithm traces a light and eye path rather than just tracing a single eye path in the unidirectional path tracing algorithm. It then weights each sampling technique along all path length combinations using multiple importance sampling (MIS). The final scene is the sum of all the images produced by each sampling technique for all path lengths.

2 Implementation

Most of the code is in the 'bdpt.cpp' and 'path.h' files. The comments in the code are helpful. I also modified the 'perspective.cpp ', 'block.cpp', and 'main.cpp' files to handle the case for when there is only one vertex on the eye subpath. Some of the important variables in the code are 'eyePath', 'lightPath', and 'resultLight'. They can all be found in the 'bdpt.cpp' file. The 'eyePath' and 'lightPath' variables will hold the main probabilities, the opposite probabilities, the intersection objects, the alpha values, and the specular flags. The 'resultLight' variable will hold the light image that will be used when there is only one vertex on the eye subpath.

The main probability that we store for each vertex on a path is the probability that we sample a certain vertex going forward. For example, if we are on the eye path and the path starts at z_0 and ends at y_0 like $z_0, z_1, ..., z_{t-1}, y_{s-1}, ..., y_0$, then the main probability stored at z_1 is the probability to sample z_1 from z_0. Likewise, the opposite probability is the probability that we sample that same vertex going backwards on the path. This means that if we are on the eye path, the opposite probability at z_1 would be the probability to sample z_1 from z_2 or whatever vertex is next. These probabilities are used to calculate the MIS weights.

The alpha values are also calculated when generating the path and the formulas can be found in the sources. We also store the intersection objects since we need the BSDF and the shading frames to calculate the main and opposite probabilities as well as the cosine factors for the geometry terms. Lastly, the specular flags for a given vertex indicates whether or not that vertex was sampled from a specular object. This is useful when we need to determine the MIS weights since the probability involves a dirac distribution.

After filling the paths with our necessary information, we need to start connecting the paths and calculate the MIS weights for each sampling technique. In the notation below, 's' represents the total number of vertices on the light subpath and 't' represents the total number of vertices on the eye subpath. We then have the following cases: '$t = 1$ and $s \geq 2$', '$t \geq 2$ and $s \geq 2$', '$t \geq 2$ and $s = 1$', and '$t \geq 2$ and $s = 0$'.

The trickiest special case that I had to handle was the '$t = 1$ and $s \geq 2$' case where there is only one vertex on the eye subpath. For this case, I created a global variable called 'resultLight' which is of the type ImageBlock. When we execute this special case, we first calculate the ray from the last light vertex to the eye and check that the ray does not intersect any objects in between. Next, we will find the pixel where that ray intersects the image plane. We use some transformation variables in 'perspective.cpp' to accomplish this.

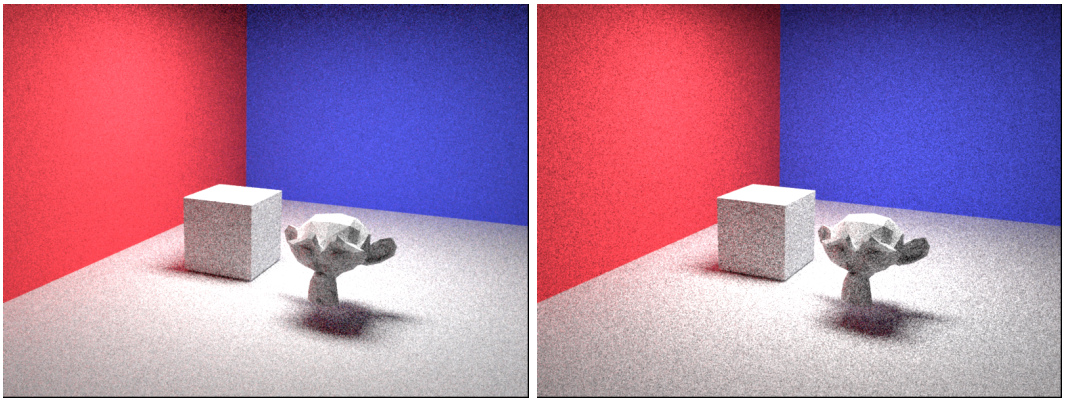
Then we calculate the color with the geometry terms, weights, alpha values, and evaluate the BSDF. Lastly, according to Veach, we divided the color that we got by the number of bidirectional samples (n) and add it to that pixel. For the other cases, I just followed what was in Veach, the PBRT book, and the Introduction to MC Rendering articles which were straightforward. The basic pseudo code for how I implemented the algorithm is below.

Algorithm 1: Bidirectional Ray Tracing

```
// Will store the light image where we will add to in the 't=1' case.
ImageBlock resultLight;
// Will store the eye vertices and light vertices that we ray traced
Path eyePath;
Path lightPath;
Fill eyePath with eye vertices (z_0, z_1, ...), alpha values, intersection structs, and specular flags using
  ray tracing and Russian Roullette;
Fill lightPath with light vertices (y_0, y_1, ...), alpha values, intersection structs, and specular flags using
  ray tracing and Russian Roullette;
// Stores the final color of the pixel in eye image after ray tracing
Color3f finalColor = Color3f(0.0f);
for t = 1 in Number of Eye Vertices do
    α_t^E = eyePath.getAlphaAt(t);
    for s = 0 in Number of Light Vertices do
        α_s^L = lightPath.getAlphaAt(s);
        if t == 1 then
            if s ≥ 2 then
                // 1 eye vertex and ≥ 2 light vertices
                Find pixel (u, v) that y_{s-1} → z_0 intersects;
                Calculate connection term c_{s,1} = G(y_{s-1} ↔ z_0) f(y_{s-1} → y_{s-1} → z_0);
                Calculate MIS weight w_{s,1};
                Add (w_{s,1} α_t^E c_{s,1} α_s^L)/n at pixel (u, v) in the light image;
        else if s == 0 then
            // ≥ 2 eye vertices and 0 light vertices
            if last eye vertex is emitter then
                Calculate connection term c_{0,t} = L_e(z_{t-1} → z_{t-2});
                Calculate MIS weight w_{0,t};
                finalColor += w_{0,t} α_t^E c_{0,t} α_s^L;
        else if s == 1 then
            // ≥ 2 eye vertices and 1 light vertices
            Calculate connection term c_{1,t} = L_e(y_0 → z_{t-1}) G(y_0 ↔ z_{t-1}) f(y_0 → z_{t-1} → z_{t-2});
            Calculate MIS weight w_{1,t};
            finalColor += w_{1,t} α_t^E c_{1,t} α_s^L;
        else
            // ≥ 2 eye vertices and ≥ 2 light vertices
            Calculate connection term
              c_{s,t} = f(y_{s-1} → z_{t-1} → z_{t-2}) G(y_{s-1} ↔ z_{t-1}) f(y_{s-2} → y_{s-1} → z_{t-1});
            Calculate MIS weight w_{s,t};
            finalColor += w_{s,t} α_t^E c_{s,t} α_s^L;
        end
    end
end
return finalColor;
```

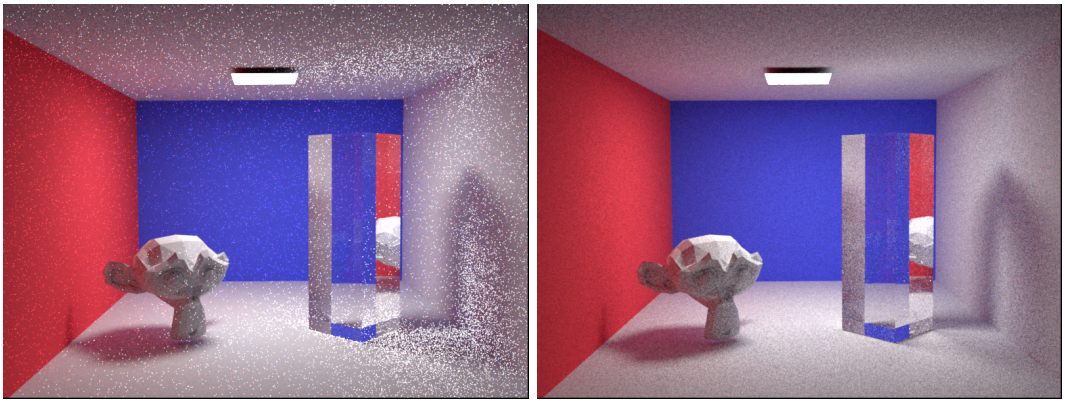
3 Experiment

We created three different images in blender. For each image, we rendered it using our '*path_mis*' ray tracer in project 4 and our bidirectional ray tracer. Each image produced by both ray tracers were rendered in about the same time. However, the number of samples that were taken were different, as described below. For most of our scenes, we can see that the bidirectional ray tracer performs better than the '*path_mis*' ray tracer. The results are below.



(a) Unidirectional ray tracing algorithm (b) Bidirectional ray tracing algorithm

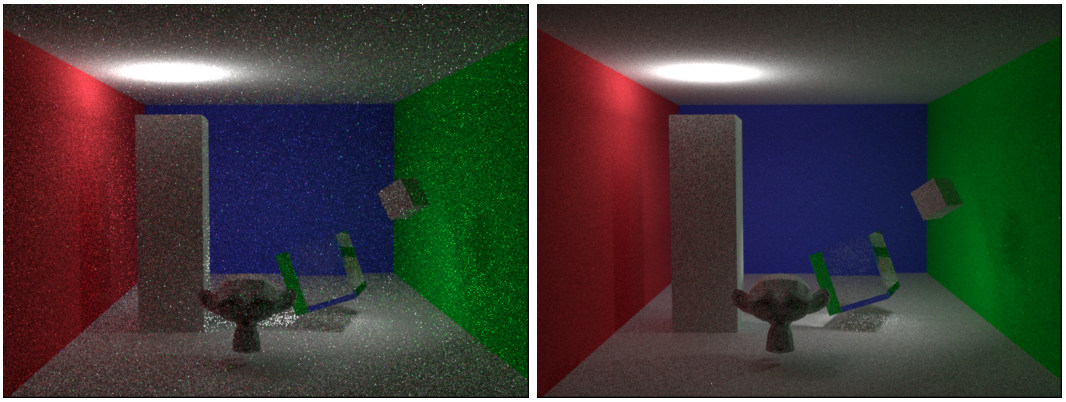
The images above displays a diffuse monkey and a diffuse cube with an area light situated on the ceiling of the box (light not shown). The scene using our '*path_mis*' ray tracer was rendered with 20 samples and took about 4.6 seconds. The scene using our bidirectional ray tracer was rendered with only 8 samples and took about 4.6 seconds as well. From the two images, there does not seem to be any difference even though there might be some more variance in the bidirectional ray tracing case. For simple scenes like this, unidirectional ray tracing might even perform better than bidirectional ray tracing. However, we can see that in the bidirectional case, only 8 samples were taken compared to 20 samples. This scene was one of the first test scenes to get the bidirectional ray tracing algorithm working.



(a) Unidirectional ray tracing algorithm (b) Bidirectional ray tracing algorithm

In our second scene, we have a diffuse monkey and a specular rectangular prism with an area light. The scene using our '*path_mis*' ray tracer was rendered with 40 samples and took about 28.0 seconds. The scene using our bidirectional ray tracer was rendered with only 8 samples and took about 27.0 seconds. We can see that there is a significant difference in the two images. There is much less variance in the bidirectional case compared to the '*path_mis*' case. With specular objects in the scene, an unidirectional ray tracing algorithm will have a lot of variance, but it will converge when a lot more samples are taken. With

bidirectional path tracing we can see that with a much smaller sample size and the same time, the scene has so much less variance.



(a) Unidirectional ray tracing algorithm (b) Bidirectional ray tracing algorithm

In our final scene above, we have a diffuse monkey, a specular box and two light sources that are hidden from the camera. One of the light sources is above the rectangular prism next to the red pane and the other light source is attached to the floating diffuse cube next to the green pane. The scene using our '*path_mis*' ray tracer was rendered with 50 samples and took about 30.4 seconds. The scene using our bidirectional path tracer was rendered with only 8 samples and took about 28.5 seconds. We can see that there is much less variance in the scene with bidirectional ray tracing than in the scene with unidirectional ray tracing. This scene shows that bidirectional ray tracing can converge much faster than unidirectional ray tracing for light sources that are not seen by the eye. With unidirectional ray tracing, the scene will converge, but it will take a lot more samples and much more time.

4 Discussion

We can see that for most of the complex scenes with hidden lights and specular objects, bidirectional path tracing converges much faster than unidirectional path tracing even though bidirectional path tracing seems to be more computationally expensive. From our experiments, given the same time for both path tracers to render a scene, bidirectional path tracing renders complex scenes with specular objects and hidden lights with much less variance than ordinary path tracing. For simpler scenes such as the first scene, there seems to not be too much difference in the images rendered. Using bidirectional ray tracing for simple scenes like this might not benefit us. In summary, we can see the power that bidirectional ray tracing has and how much faster it can converge than unidirectional ray tracing for complex images with specular objects and hidden lights.

5 Time Spent and Side Note

Overall, I spent at least 50 hours on debugging, coding, and researching the topic. On a side note, when you run the code, the image that is being rendered and displayed in the gui is just the eye image. The 'correct' image with the light image and eye image added together will be saved to the file after the rendering is done.

6 Sources

Veach, Eric. (1997). Bidirectional Path Tracing. $http://graphics.stanford.edu/papers/veach\_thesis/$.

Pharr M., Wenzel J., & Humphreys G. (2004). Physically Based Rendering: From Theory To Implementation. $http://www.pbr-book.org/3ed-2018/Light\_Transport\_III\_Bidirectional\_Methods/Bidirectional\_Path\_Tracing.html$

Zwicker, M. (n.d.). Monte Carlo Rendering: A Brief Introduction. Class Handout.