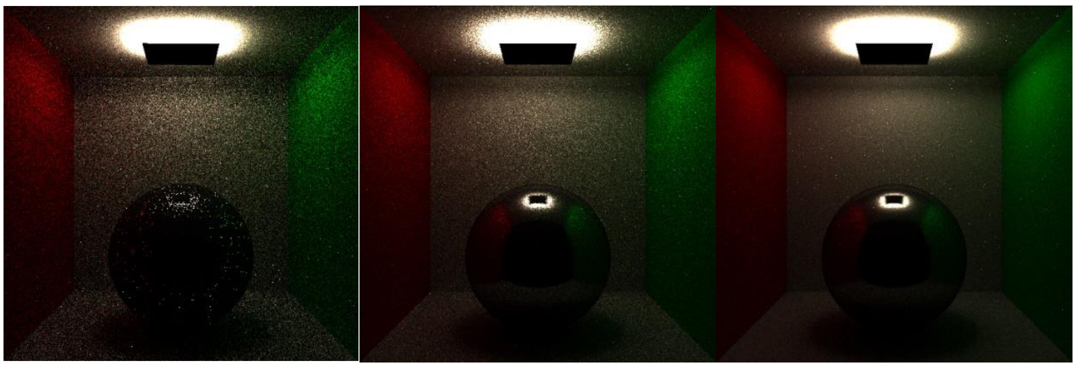
CSE168 Final Project
Bidirectional Path Tracing and Multiplexed Metropolis Light Transport

Zeyu Wang



In the final project, I implemented the bidirectional path tracing algorithm [1] and multiplexed Metropolis light transport [2] in the OptiX 6.5 renderer developed in the course.

Bidirectional Path Tracing (BDPT)

In conventional path tracing, rays are generated from the camera and then traverse through the scene. However, this scheme will become inefficient when the lighting situation becomes complex. For example, if the light is facing towards a wall with a little margin, rays from the camera can only reach the light source with a low possibility, causing a high-variance result.

To alleviate this issue, BDPT traces two sub-paths from both the camera and light sources. The camera path is transporting importance, and the light path is transporting radiance. When the BRDF is symmetric, there is no need to differentiate them. Each sub-path contains a *maxDepth+1* number of path vertices. The vertices record the information of the path such as position or surface normal. Then, the two sub-paths are connected on each possible vertices pair and form a group of full paths. The radiance a path is carrying can therefore be computed from the information recorded on the end vertices of the sub-paths.

For one specific path, there are also multiple possible connection strategies. For example, a path with 3 vertices can be formed by 3 vertices from the camera sub-path and 0 vertex from the light sub-path (similar to standard path tracing), or 2 from the camera and 1 from the light (similar to next event estimation), or 1 from the camera and 2 from the light. To better combine these strategies, multiple importance sampling can be adopted. The power heuristic is applied to weight the radiance of the path by its forward and reverse possibility.

Multiplexed Metropolis Light Transport (MMLT)

Metropolis Light Transport (MLT) [3] is an entirely different rendering techniques than Monte Carlo path tracing. Leveraging Metropolis Hastings sampling, MLT becomes a Markov Chain Monte Carlo method. It first selects a group of paths as the starting point of each Markov chain. Then, it iteratively proposes a new path by either perturbing it slightly or generating a new path. The proposed new path is either accepted or rejected as the new seed path according to the weight of the paths. The result finally converges to the unbiased rendering by multiplying with a normalization factor, which can be computed from a bootstrap phase with path tracing or BDPT. It explores more in the important areas of the path space thus improving efficiency.

However, the original MLT algorithm is very complicated to implement since it needs to modify a path in the path space. Primary Sample Space Metropolis Light Transport (PSSSMLT) [4] tackles this problem by stating that the paths are generated from a sequence of random numbers (primary samples), then if the random numbers are mutated, the paths are mutated as well. Multiplexed Metropolis Light Transport (MMLT) [2] further refines the focus of initial path mutation to paths of each specific different depths instead of the camera or light sub-paths. The depth value is fixed during mutation and many chains are launched to render the final results. The consequence is that more computation is spent around paths with large MIS weight in BDPT bootstrap so that the sampling is even more efficient.

Implementation Details

Implementation:

The majorant part of codes is written in the file *PathMethod.cu*. There are also some additional functions written in other files. *BRDF.cu* contains codes related to BRDF, and *Sampler.cu* implements two samplers to generate random samples. Slight modifications are also made to the existing files in the framework, e.g., a PCG32 random number generator is appended to *random.h.* The implementation is highly inspired by PBRT-V4 [5]. The book and its code repository provide detailed explanations on the topics. Since my renderer does not incorporate refractive BRDFs and volume path tracing, many simplifications are made.

Dynamic Allocation:

It might seem necessary to use dynamic memory allocation in the device code, for example, vertices array in sub-path generation. However, since the maximum size of the required memory is known in advance, I can always allocate enough space from the host side once and before launching the device code with a tradeoff of extra memory consumption, therefore avoiding dynamic allocation in device code. For example, the sub-path may contain at most *maxDepth+1* number of vertices and there are in total *width*height* threads. In this case, a buffer object of *Vertex* type of size *width*height*(maxDepth+1)* is allocated from the host side and passed to the device code.

Race Condition:

In the original provided framework, rays are assumed to be generated for one specific pixel. However, this property no longer holds for BDPT and MMLT. Race condition may occur when multiple threads try to write to the same position in the result buffer. Unfortunately, atomic operations are not supported in OptiX 6.5. Thus, race condition cannot be avoided unless an entire modification to the framework is made. As an alternative, two versions of code are written and are switched by a macro *RACE* defined on the top of *PathMethod.h*.

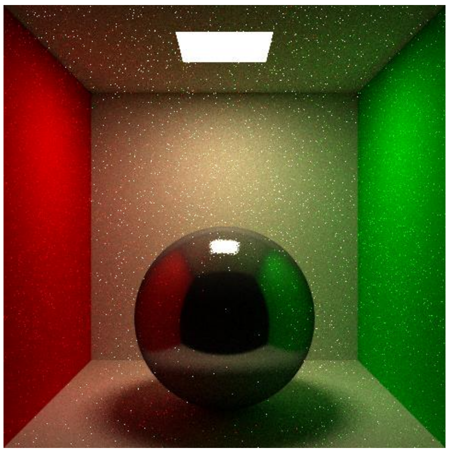
When the macro is not defined, the code is race-free. BDPT now only supports camera paths with a minimum of two vertices. In this scenario, no matter how full paths are connected, the position of the ray on the camera film never changes. One thread only writes to one pixel. On the other hand, MMLT no longer works correctly as the paths are confined in one pixel. To mimic the concept of MMLT, the number of mutations of each pixel is set proportional to the total weight of all bootstrap samples in that pixel.

When the macro is defined, the code is race risky. BDPT now supports camera paths with a minimum of one vertex, meaning that if connecting it to a light sub-path vertex, the resulting point on the camera film might be different from the pixel the thread corresponds to. There is chance that the radiance from some thread is overwritten due to race condition. MMLT is then the correct implementation.

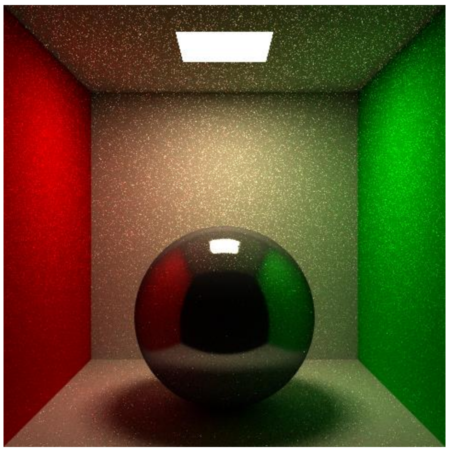
Results:

Scene: cornellBRDF.test

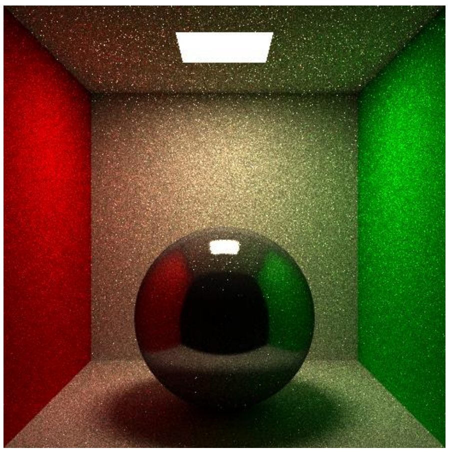
Path Tracing



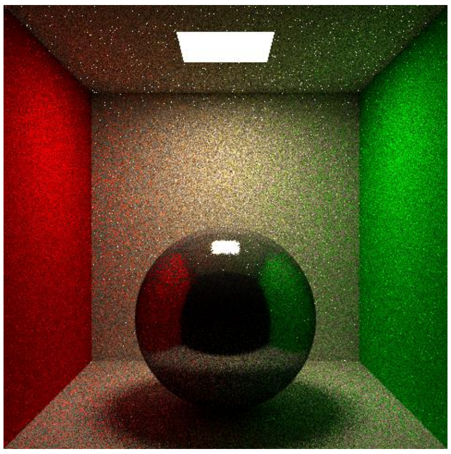
BDPT No Race



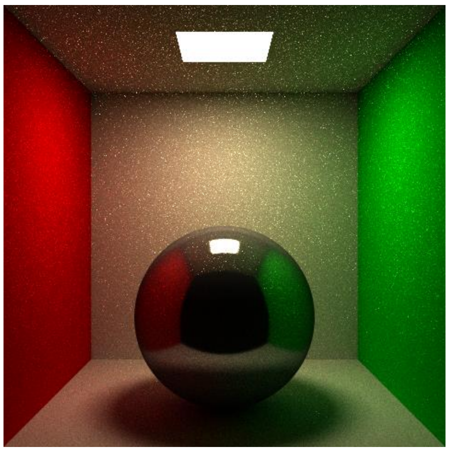
BDPT Race



MMLT No Race



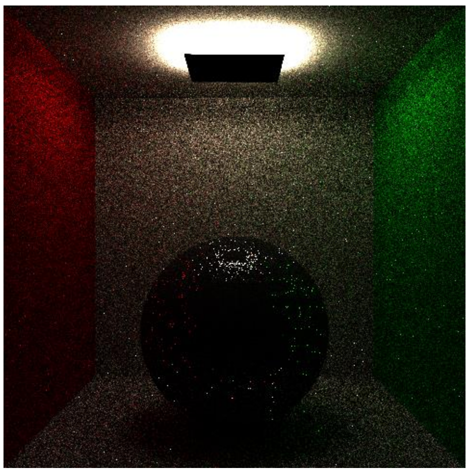
MMLT Race



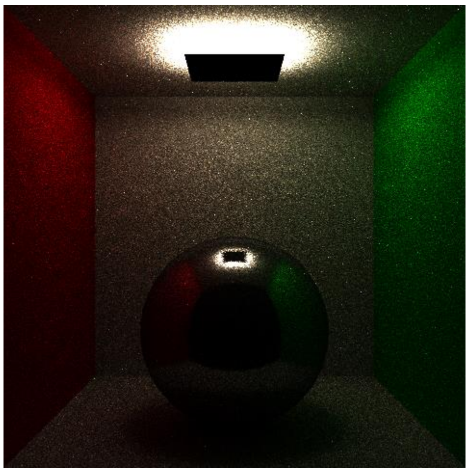
This reveals that the implementations are correct and unbiased. Surprisingly, race condition does not have a large impact on the final results.

Scene: cornellOcclusion.test. BDPT and MMLT are set to RACE.

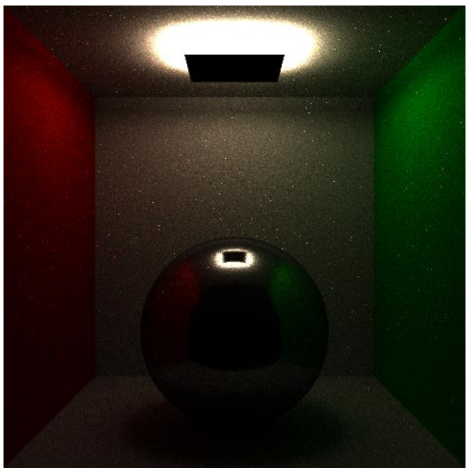
Path Tracing



BDPT



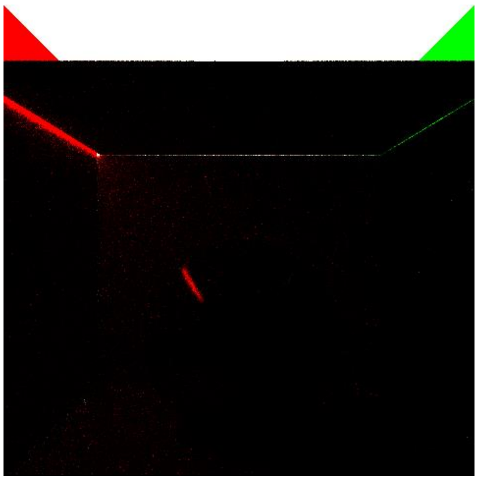
MMLT



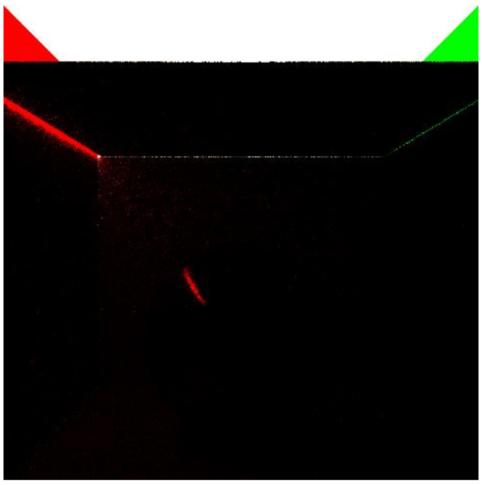
This scene shows the superior efficiency of MMLT and BDPT than path tracing.

Scene: cornellSlab.test BDPT and MMLT are set to RACE.

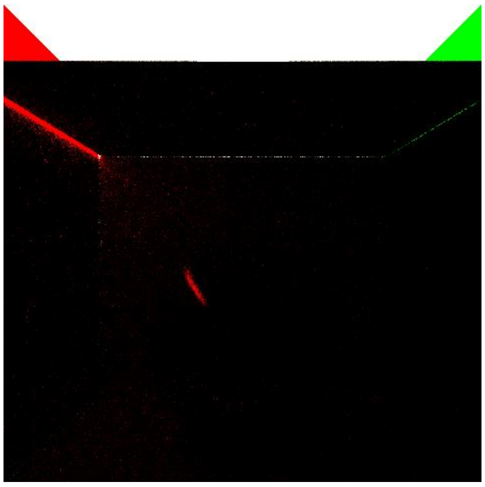
Path Tracing



BDPT



MMLT



This scene shows that MMLT can better explore the local path space. The red color is spread to a larger area.

Conclusion

In this project, I successfully implemented two modern rendering algorithms, respectively, bidirectional path tracing and multiplexed Metropolis light transport. The examples show that they are more efficient in sampling the paths, resulting in less noisy images. I did not compare the running time because fundamentally they are implemented differently. BDPT is highly parallelable while MMLT is only parallel for a much smaller number of Markov chains. Also, MMLT requires two separate kernel launches for the bootstrap and the mutation, which makes it more unclear about how to compare the running time.

References

[1] Eric P. Lafortune, and Yves D. Willems. 1993. Bi-Directional Path Tracing. Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (Compugraphics '93). https://graphics.cs.kuleuven.be/publications/BDPT/BDPT_paper.pdf

[2] Toshiya Hachisuka, Anton S. Kaplanyan, and Carsten Dachsbacher. 2014. Multiplexed metropolis light transport. ACM Trans. Graph. 33, 4, Article 100 (July 2014), 10 pages. https://doi.org/10.1145/2601097.2601138

[3] Eric Veach, and Leonidas J. Guibas. 1997. Metropolis Light Transport. SIGGRAPH 97 Proceedings (August 2018), pp. 65-76. https://graphics.stanford.edu/papers/metro/metro.pdf

[4] Csaba Kelemen, László Szirmay-Kalos, György Antal, and Ferenc Csonka. 2022. A Simple and Robust Mutation Strategy for the Metropolis Light Transport Algorithm. Computer Graphics Forum 2002. Volume 21. http://cg.iit.bme.hu/~szirmay/paper50_electronic.pdf

[5] Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2018. Physically Based Rendering: From Theory To Implementation. Version 3. https://pbr-book.org/