15462/15662 Computer Graphics

Assignment #5

Yu Mao (ymao1), Rui Zhu (rz1)
Robotics Institute, Carnegie Mellon University

December 17, 2015

Contents

1. Overview **2**

2. Implemented Tasks **3**
2.1 Task 1: Parallel BVH construction . . . 3
2.1.1 Morton-code based BVH construction . . . 3
2.1.2 Parallel Binary Radix Tree construction . . . 4
2.1.3 CUDA-accelerated BVH construction . . . 5
2.1.4 Implementation . . . 5
2.1.5 How to run a demo . . . 5
2.1.6 Evaluation . . . 6
2.2 Task 2: Bidirectional Path Tracing (BDPT) . . . 7
2.2.1 Motivation . . . 7
2.2.2 Approach . . . 7
2.2.3 Implementation . . . 8
2.2.4 How to run a demo . . . 9
2.2.5 Evaluation . . . 9

3. Challenges and Future Work **15**
3.1 CUDA programming . . . 15
3.2 The BDPT task . . . 15

Bibliography **15**

1. Overview

We completed two tasks within this project: **Parallel BVH Construction with CUDA Acceleration**, and **Bidirectional Path Tracing**.

For the first task, we first implemented the Morton code based method which only runs on CPU. Then we implemented the parallel binary radix tree(BRT) construction algorithm as proposed in [2] for general BRT construction. Thirdly, we implemented the parallel algorithm for BVH construction based on BRT construction algorithm, which augments it by adding a bottom-up parallel bounding box construction phase. We also compared the efficiency of different approaches.

For the second one, we modified the renderer used in Homework 3, which uses traditional path tracing, to one which uses Bidirectional Path Tracing (BDPT). The new renderer is able to render scenes with AreaLight using BDPT with better illumination and variation in shadowed areas. We compared the efficiency of there two methods, as well as the “photorealisticity" of the rendered images under these two methods. Our BDPT gives better illumination in the soft shaded area, and less variance in the caustic focused light under a glass sphere.

This is a really cool screen record of our BDPT renderer: https://youtu.be/gRIJggimCf4

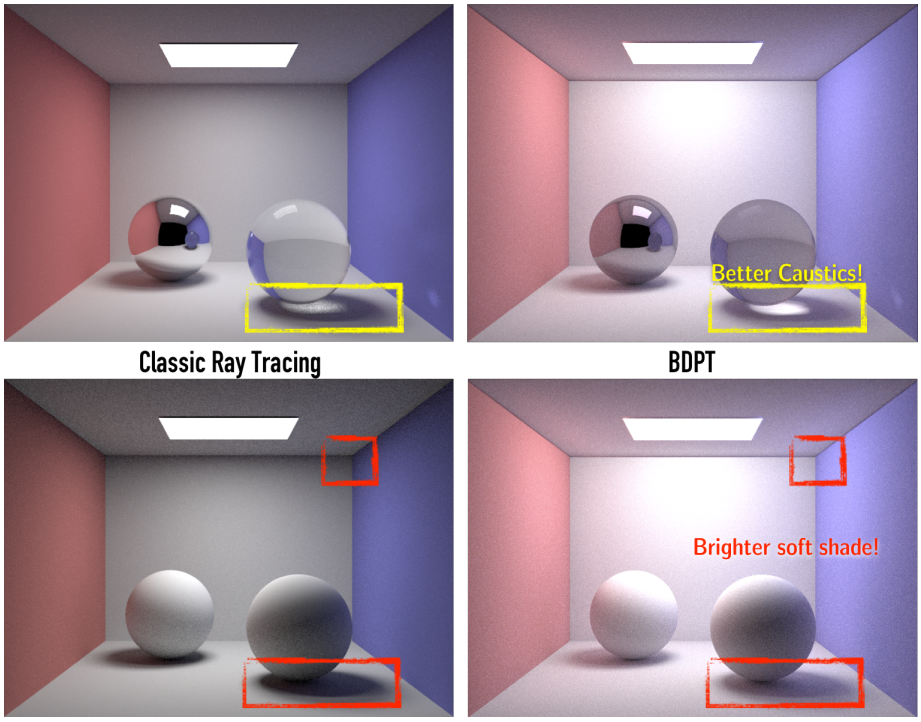

Figure 1: Featurs of BDPT

2. Implemented Tasks

2.1 Task 1: Parallel BVH construction

2.1.1 Morton-code based BVH construction

A well-known approach to accelerate the BVH construction process is to use Morton code. Morton code for a given point whose position is defined in unit cube is given as $X_0Y_0Z_0X_1Y_1Z_1...$ where the x coordinate of a point is $0.X_0X_1X_2...$, y coordinate is $0.Y_0Y_1Y_2...$ and z coordinate is $0.Z_0Z_1Z_2....$ An example is illustrated in Figures 2.

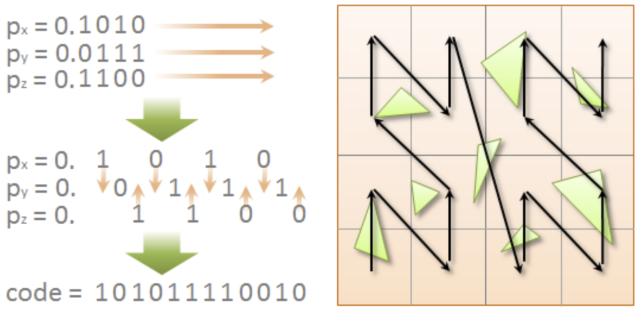

Figure 2: Morton Code Example

As proposed in [4] We first calculate the Morton code for each primitives according to their center point. Then we sort the primitives using the Morton code, after which the primitives will also be placed in a spatially coherent way. Then we construct the BVH by splitting the range of the current root node recursively. Specifically, for the current root node which covers range$[i, j]$, we find the highest bit that differes between the Morton code in range $[i, j]$, noted as γ. Then the left child of the current node spans $[i, \gamma]$, and the right child spans $[\gamma + 1, j]$. And we need to perform this step recursively until the current node spans only a single primitive. Then we are done.

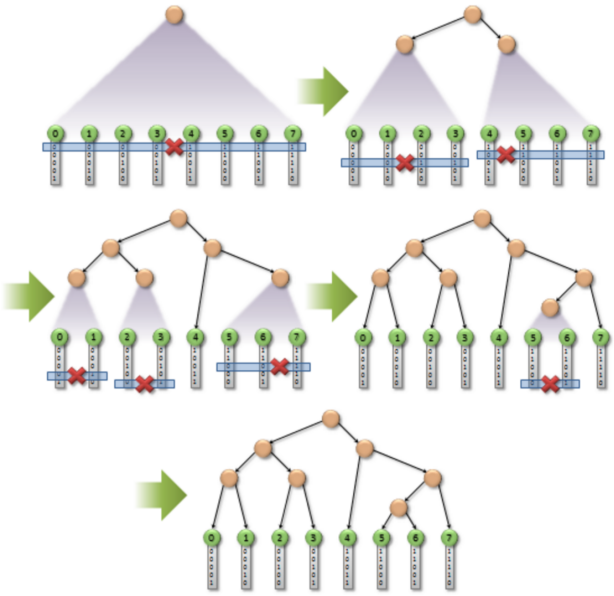

Figure 3: Morton Code Example

2.1.2 Parallel Binary Radix Tree construction

Morton code is awesome because it gives us a boost in BVH construction speed. But to fully utilize the calculating capacity of modern GPU, we need to think about parallelizing this algorithm. It is not hard to notice that the hierachy generation process is a binary radix tree construction process.

Given a set of n keys $k0,...,kn$ represented as bit strings, a binary radix tree (also called a Patricia tree) is a hierarchical representation of their common prefixes. The keys are represented by the leaf nodes, and each internal node corresponds to the longest common prefix shared by the keys in its respective sub-trees.

We follow the idea of [2] and implemented the parallel binary radix tree builder which runs on CUDA. In our implementation, each threads would process one internal node. The key part of this algorithm is the layout of nodes. Each node is located at I_0 and indices of its childeren are assigned according to its respective split position. The left child is located at I_γ if it covers more than one key, or at L_γ if it is a leaf. The rule applies to the right children as well. An example is illustrated in Figures 4. Using this particular layout, we can easily infer the range of each node. Then parallel internal construction node is feasible.

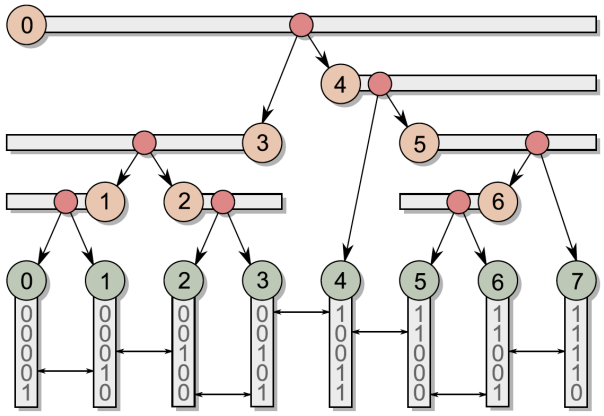

Figure 4: Node Layout for parallel binary tree construction

2.1.3 CUDA-accelerated BVH construction

After constructing the Binary Radix Tree, we need also to construct the bounding box of each node. The conventional top-down approach could be very slow and fails to utilize the intermediate result. Thus, we adopted the parallel algorithm proposed in [2] by starting from the bottom. In this algorithm, each path from leaf node to root are processed in parallel. For each internal node, we use $atomic_counter$ to record how many threads have visited this node. The first thread visited the node would be terminated, i.e., the thread number decreases by half after finished processing the current level. This ensures that every node gets processed only once, and not before both of its children are processed.

2.1.4 Implementation

Based on the reference project of assignment 3, we implemented CPU Morton Code method in $bvh.cpp$, GPU Morton Code method in $parallelBRTreeBuilder.cu$. The specific BVH method to be used can be configured in the first lines of $bvh.cpp$ by enabling the corresponding Macros.

We uploaded a youtube video to show how fast our BVH construction process is (https://youtu.be/MotzB-XGoaE).

2.1.5 How to run a demo

Before compiling the program, make sure you have:

1. CUDA compatible Graphics Card

2. CUDA SDK installed

```
./pathtracer ../dae/sky/CBcoil.dae
```

After the scene is loaded, hit 'v' to start BVH construction.

2.1.6 Evaluation

We compared the performance of three algorithms on three models (cow.dae, CBbunny.dae and CBcoil.dae) which can be found under the project folder. As the result shows, the Morton code method performs way faster than the SAH-based method. Morton code method with CUDA acceleration is even faster.

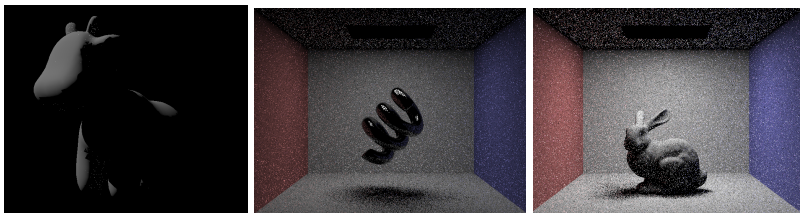

Figure 5: Input Model File

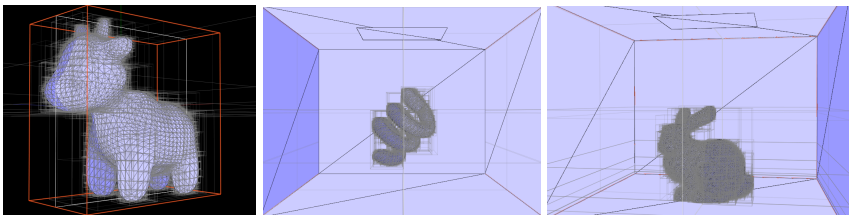

Figure 6: BVHs constructed

Table 1: BVH Construction time(second) with Different Methods

	SAH(Default)	Morton Code(CPU)	Morton Code(GPU)
cow.dae	0.0218	0.0047	0.0043
CBcoil.dae	0.0320	0.0075	0.0057
CBbunny.dae	0.1446	0.0325	0.0183

However, We should also consider the quality of the constructed BVH. For simplicity, we will use the rendering time to measure the BVH quality. As the result shows below,

Table 2: Rendering Time(second) with BVHs Constructed from Different Methods

	SAH(Default)	Morton Code(CPU)	Morton Code(GPU)
cow.dae	0.7827	0.7896	0.7670
CBcoil.dae	1.3357	1.3944	2.3644
CBbunny.dae	1.3202	1.4370	2.5534

BVH constructed using SAH-based method has the best quality, which makes sense because Morton code is an approximation method.

2.2 Task 2: Bidirectional Path Tracing (BDPT)

2.2.1 Motivation

Bidirectional Path Tracing (BDPT) was originally developed by [3]. The main idea is to generate two paths from the eye (pixels to the camera pinhole) and the light sources respectively, and accumulating contributions from the transport paths inter-wined by points in these two paths to pixels on the photo.

BDPT is especially suitable for rendering scenes where lights only directly light up small part of the scene, or huge shadows are observable in the scene. In this cases global illumination rendering with BDPT has better details, lighting as well as reality in shadowed areas compared to traditional path tracing methods in that the light path carries the luminance from the light source to the scene, hopefully the shadowed areas, rather than passively waiting for the eye rays ending up in the light source one day.

For this sub-task, our goal is to modify the renderer used in Homework 3, which uses traditional path tracing, to one which uses BDPT. We will compare the efficiency of there two methods, as well as the “photorealisticity" of the rendered images under these two methods.

2.2.2 Approach

Figure 7 is an overview of the bi-pathes in BDPT from the master thesis [1]. The two paths, namely eye path and light path are generated respectively by random walk from a camera ray corresponding to a certain pixel in the photo, and a sample light ray. The random walk ends if the next expected ray in the path is absorbed or if the depth of the path exceeds a certain threshold. Both radiance and importance (weight of a path) are traced and recorded for later combination.

Then, we will need to evaluate the contribution of a path, which starts from the light, travels along the light path for a while, and jump to the eye path, and eventually end up to

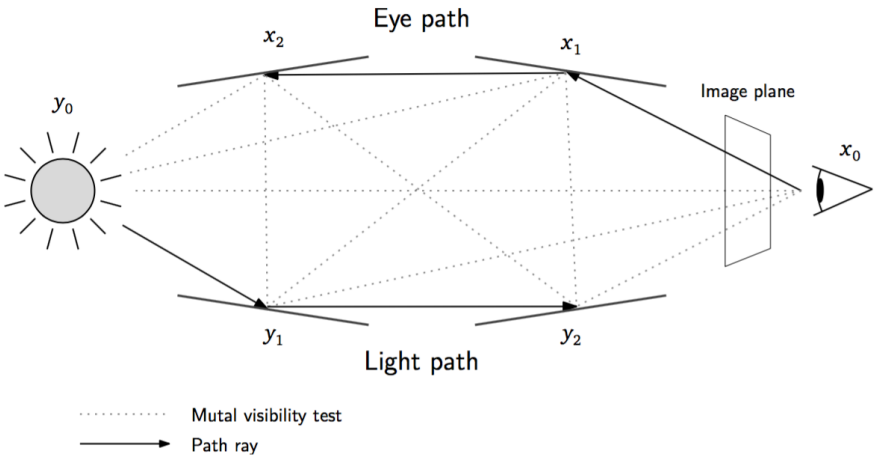

Figure 7: The BDPT Overview

a pixel on the photo. We denote the number of vertices the path travels along the eye path as S, and the number of vertices the path travels along the light path as T. Lafortune [3] differentiates between four cases of the combination of S and T, and suggests we evaluate the contributions of the four cases respectively.

1. $S = 0, T > 0$. Direct ray from light to eye;
2. $S > 0, T = 0$. Classic Ray Tracing;
3. $S = 0, T > 0$. Light travels along light path and then goes directly to eye;
4. $S > 0, T > 0$. Bi-Path.

Details on how to evaluate contributions in the four cases, including how to assign weights to ensure the total contribution converge to the ground truth can be found in Lafortune's origional paper [3] and a nice summing up in Jakob's master thesis [1].

2.2.3 Implementation

To midify the path tracer of the Homework 3 to use BDPT, the origional path-tracing function `PathTracer::trace_ray` has to be replace by a new function `Spectrum PathTracer::trace_ray_bpt` to "trace" a camera bidirectionally. In this function, a light ray is sampled, and the light paths and eye paths are generated by random walk with function `PathTracer::randomWalk`, and then contributions in the four cases aforementioned are calculated and added to the photo simultaneously.

It is worthwhile to mention that in Case III, the contribution will possibly not add to the pixel where the camera ray comes from, because in this case, rays from the light path directly shoots into the camera pinhole from possible positions all over the scene. As a result, apart from the wave-like rendered tiles gradually piled up from the bottom of the image, you may hopefully see the whole picture is gradually lit up by light rays landing on the pixels of the image.

2.2.4 How to run a demo

```
./pathtracer -t 18 -s 16 -p 1 ../dae/sky/CBspheres.dae
```

Notes:

The `-p` parameter is a switch for BDPT or classic path tracing (1 is BDPT; 0 for classic path tracing (default)).

To sample light rays from a light source, we need to implement functions named `SceneLight::sampleLight` and `SceneLight::sampleLightFromP` for every light class. Since I intend mainly to focus on the new BDPT renderer, I only implemented these two functions for `AreaLight`. As a result, only scenes using this type of light, e.g. **the Cornell Box series**, will render correctly with BDPT.

2.2.5 Evaluation

We got a reference code for HW3 from Bryce, and we made comparisons between classic path tracing using this code, and our code for BDPT. We tested with a picture resolution of 960*640, 256 camera rays per pixel and 18 threads.

For the **CBspheres_lambertian** case, the picture rendered by classic path tracing (see Figure. 8):

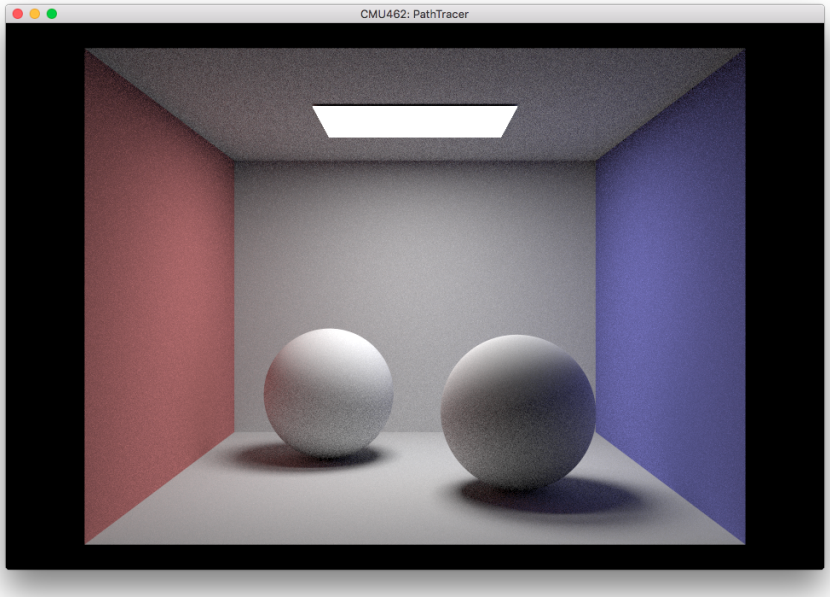

Figure 8: CBsphere_lambertian rendered with HW3 reference code

Total time consumed: 195.1294s
Image rendered by our BDPT renderer (see Figure. 9) :

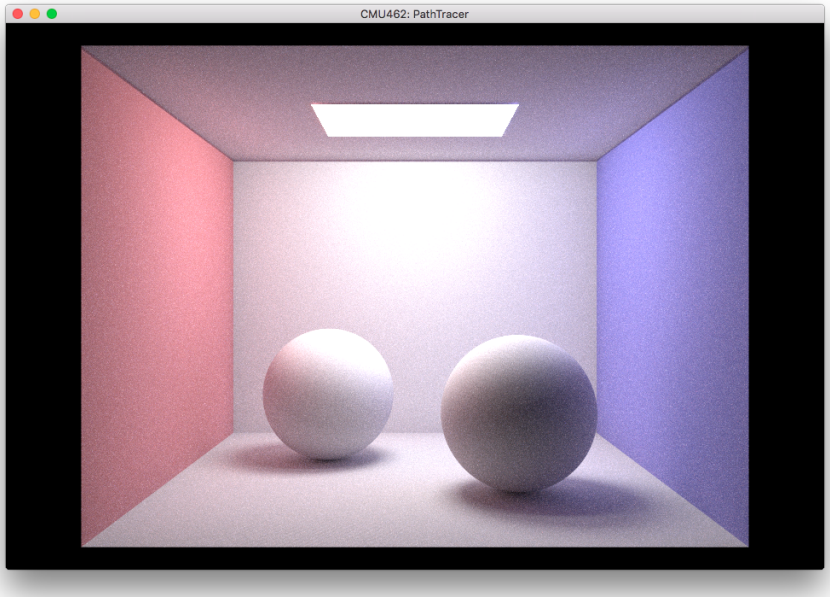

Figure 9: CBsphere_lambertian rendered with our BDPT renderer

Total time consumed: 2353.5625s

We may easily observe that, the BDPT rendered is better at rendering shades, in that the two **shades** in our rendered images is better illuminated that the classic path tracing. Meanwhile, our renderer takes significantly more time (almost 10X) than the classic one. We may need to adopt methods like Russian Roulette, GPU acceleration, or simply cut the eye and light paths for better efficiency (the computation grows exponentially with the two path length).

For the **CBspheres** case, the reference render will not render the glass sphere correctly, so we take Sky's reference picture on the HW3 homepage for comparison. The output image from the classic path tracing (see Figure. 10) :

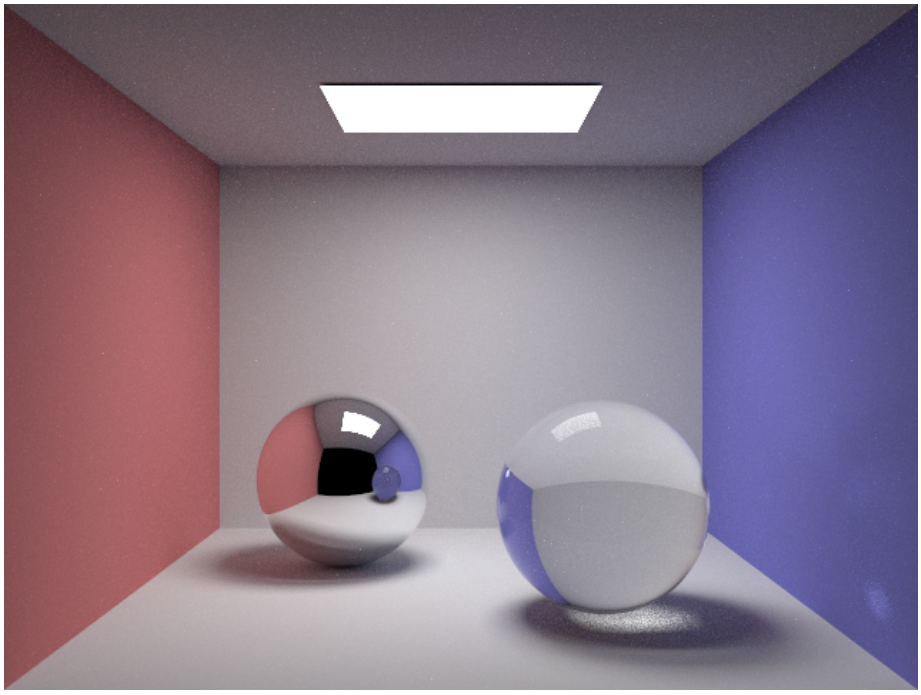

Figure 10: CBsphere rendered by Sky

Total time consumed (on Sky's machine): 423.7795s
Image rendered by our BDPT renderer (see Figure. 11) :

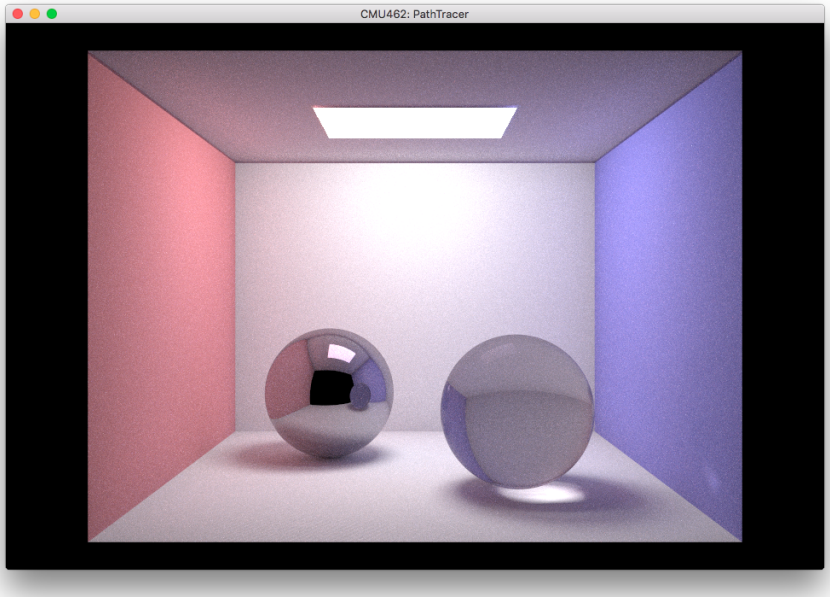

Figure 11: CBsphere rendered with our BDPT renderer

This is a screenshot when rendering is in progress.

Total time consumed: 2764.4224s

In the BDPT rendered image, we observe that the image is somehow dimmer as compared with the classic ray tracing case. We are still tracing the bug (possibly, because we do not know the ground truth to determine if it is a bug or it is exactly what a natural scene should be) and will update this Github page and repository if we fix the bug in the future. **(To whoever is reading this: We would very much appreciate it if you had any guess and discuss with us about this issue).** Nevertheless, the BDPT rendered image shows significantly **less variance** in the caustic focused light under the glass sphere, as well as better lit-up shading.

And finally a screenshot of render in progrress (see Figure 12; notice the wave-like queues of tiles and pixels being rendered, and the gradually lit-up background due to Case III)

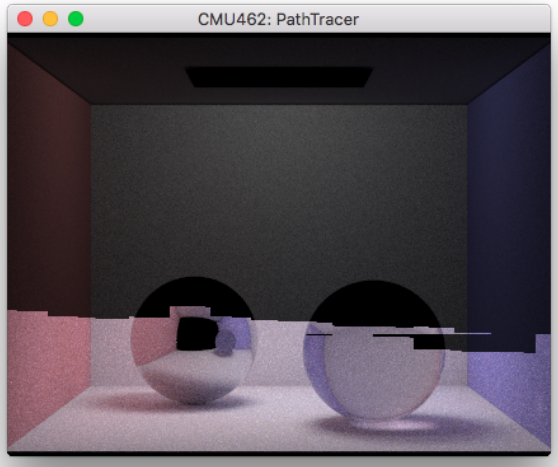

Figure 12: Render in prograss

This is a really cool screen record of our renderer: https://youtu.be/gRIJggimCf4

3. Challenges and Future Work

3.1 CUDA programming

Since neither of us has any experiences using CUDA. We spent a lot of time on it. The first thing is environment setup, because we do not have NVIDIA graphics card installed in our computer, we have to use the ones in GHC, which is sadly, very popular. After that we start programming using CUDA and we found that debugging parallel code is notoriously hard. We have to perform unit test to every component before integrating them, which is a very unique experience.

3.2 The BDPT task

For the BDPT task, the problem which troubled us for a long time as the `EPS_D` bias item in intersection test. Because intersection tests are required among light samples and scene points, proper intersection tests are importent. We later found out that, adding the `EPS_D * direction` to shadow ray is critical for successful test and numerical reasons, and improperly large or small `EPS_D` will cause total black glass sphere or black rings in it.

Future work may include: adding support for more light sources, accelerating with CUDA (inspired by our work in the first task, and the parallelism within BDPT).

Bibliography

[1] Wenzel Jakob. "Accelerating the bidirectional path tracing algorithm using a dedicated intersection processor". MA thesis. Karlsruhe Institute of Technology (KIT), July 2007.

[2] Tero Karras. "Maximizing Parallelism in the Construction of BVHs, Octrees, and K-d Trees". In: *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*. EGGH-HPG'12. Paris, France: Eurographics Association, 2012, pp. 33–37. ISBN: 978-3-905674-41-5. DOI: 10.2312/EGGH/HPG12/033-037. URL: http://dx.doi.org/10.2312/EGGH/HPG12/033-037.

[3] Eric P Lafortune and Yves D Willems. "Bi-directional path tracing". In: *Proceedings of CompuGraphics*. Vol. 93. 1993, pp. 145–153.

[4] C. Lauterbach et al. "Fast bvh construction on gpus". In: *IN PROC. EUROGRAPHICS '09*. 2009.