CSC4140 Assignment 6

Computer Graphics

April 22, 2022

Ray Tracing

This assignment is 10% of the total mark.

Strict Due Date: 11:59PM, Apr 22^{th}th , 2022

Student ID: 119010344

Student Name: Xiao Nan

This assignment represents my own work in accordance with University regulations.

Signature: Xiao Nan

Contents

1 Overview

In this assignment I implemented a basic pipeline on ray tracing. Specifically in part one, I implemented the ray generation, ray-triangle and ray-sphere intersaction tests and the pixel illumination. In part two I further implemented the BVH construction and intersaction tests to accelerate the ray-scene intersaction testing process. In the third part I implement the direct illumination for the pixel point using the Monte-Carlo estimate for solving the rendering equation, then in the forth part I implemented the indirect illumination, a recursive version of illumination for rendering the lights with different bounces. Finally in the fifth part, I implement the adaptive sampling for the rays given a pixel in order to optimize the time comsumption for the ray sampling procedure.

2 Ray generation and intersaction tests

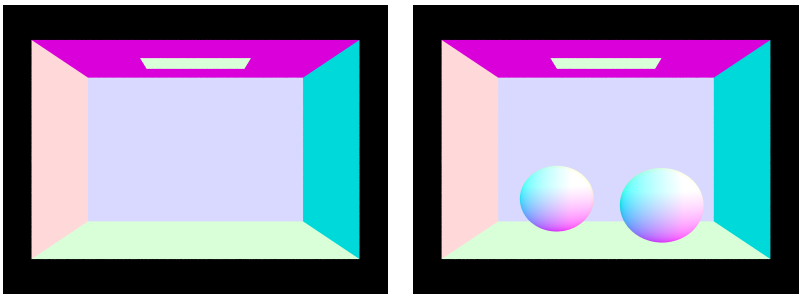
2.1 Principles of ray generation and intersaction tests (with triangle and sphere)

Ray generation refers to the process of generating a number of rays, originated from the camera and specified by the maximum number given by the command line argument settings, for each pixel under the screen space in order to simulate the illumination effects on that pixel from the incoming lights from different directions. Specifically, the incoming rays are grid-sampled (uniformly sampling from the unit square for the hit point's offset from the integer coordinates) and their illuminance are averaged in order to obtain the pixel's illumination value. After ray generation and BVH intersaction checkings, the rays are tested for intersaction with the primitives (Specifically, triangles and spheres) in order to know whether the light comes from the object's reflection or the light source's emission.

The triangle intersaction tests are performed based on the non-negativity of the Barycentric coordinates of the intersaction point, which is computed by the Moller Trumbore Algorithm for the faster speed of computation, and the sphere intersaction tests are implemented based on the discriminant of the quadratic equation (substituting the variables for the sphere equation with the parametrized line equation).

2.2 Results for normal shading

The results for normal shadings of *CBempty.dae* and *CBspheres-lambertian.dae* are shown in figure 1.



(a) Empty room (b) Room with sphere

Figure 1: Results for normal shadings for the room with and without balls

3 BVH construction and intersaction-test optimizations

3.1 BVH construction algorithm, and splitting heuristics

The Bounding Volume Hierarchy (BVH) algorithm is widely used in Computer Graphics to reduce the time for the ray-scene objects' intersaction testing. It builds a binary tree with each internal node containing the information of a bounding box of a part of the scene, and the leaf nodes containing the information of the primitives within the leaf bounding box. In this way, the ray-primitive intersaction test can be performed firstly on the bounding boxes in the internal nodes of the BVH tree by using a binary search, then finally on the leaf node where each primitive is tested for intersaction. Such algorithm reduces the time complexity to **O(nlog(n))** instead of the original **O(n)**.

I split the primitives by sorting the primitives with respect to a given axis (x, y or z) and half-splitting the resulting vector: The axis to split is chosen in sequence with x, y and z (that is, a global **num-of-recur** is recorded and the axis to split depends on $\frac{\textbf{num-of-recur}}{3}$). For a given axis, I sort the primitive pointer array according to the axis's values of the primitive's centroid (e.g. **x** value of the centroids of each primitive), and then split the resulting sorted array (permuted according to an axis) in half.

3.2 Results for rendering with and without BVH acceleration

After the BVH acceleration, some complex and large images, such as the *maxplanck.dae*, are successfully rendered with normal shading, as shown in figure 2, which demonstrates the strong ability of acceleration of the BVH algorithm.

Figure 2: Successfully render Max-Plank with BVH

The resulting rendering times, with and without BVH acceleration, for rendering the *teapot.dae* are shown in figure 3. We may see that the number of primitives to be rendered is significantly increased without using the BVH acceleration algorithm, and hence the rendering time is almost **35** times longer than that with using the BVH acceleration algorithm, which further shows the BVH algorithm's power on rendering median or large-sized objects.

(a) The image to be rendered

```
graphics@Graphics-CUHKSZ:~/Desktop/HW6/rt$ ./pathtracer -r 2400 1800 -f cow.png dae/meshedit/teapot.dae
[PathTracer] Input scene file: dae/meshedit/teapot.dae
[PathTracer] Rendering using 1 threads
[PathTracer] Collecting primitives... Done! (0.0006 sec)
[PathTracer] Building BVH from 2464 primitives... Done! (0.0239 sec)
[PathTracer] Rendering... 100%! (16.1789s)
[PathTracer] BVH traced 4320000 rays.
[PathTracer] Average speed 0.2670 million rays per second.
[PathTracer] Averaged 6.278985 intersection tests per ray.
[PathTracer] Saving to file: cow.png... Done!
[PathTracer] Job completed.
```

(b) Rendering time with BVH acceleration

```
graphics@Graphics-CUHKSZ:~/Desktop/HW6/rt$ ./pathtracer -r 2400 1800 -f cow.png dae/meshedit/teapot.dae
[PathTracer] Input scene file: dae/meshedit/teapot.dae
[PathTracer] Rendering using 1 threads
[PathTracer] Collecting primitives... Done! (0.0007 sec)
[PathTracer] Building BVH from 2464 primitives... Done! (0.0287 sec)
[PathTracer] Rendering... 100%! (709.9340s)
[PathTracer] BVH traced 4320000 rays.
[PathTracer] Average speed 0.0061 million rays per second.
[PathTracer] Averaged 2464.000000 intersection tests per ray.
[PathTracer] Saving to file: cow.png... Done!
[PathTracer] Job completed.
```

(c) Rendering time without BVH acceleration

Figure 3: Results for normal shadings with and without BVH acceleration

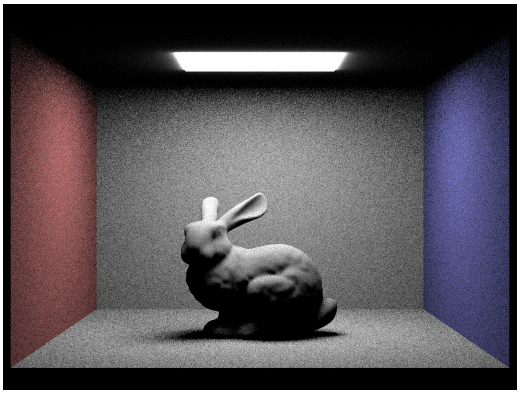
4 Direct illumination

4.1 Direct illumination with uniform hemisphere sampling and importance sampling

The illumination, consisting of the zero-bounce illumination, direct illumination and indirect illumination, describes the illuminance of a given pixel affected by the incoming lights. The rendering equation made it possible to estimate such illumination using the Monte-Carlo integration on the incoming illuminance given by the sampled incoming lights, as well as the **BSDF**, the material reflectional properties that determines the reflectance of the light towards a given outgoing light direction. In order to sample those incoming lights, we may either use the uniform hemisphere sampling method, sampling from the unit hemisphere uniformly for the incoming light's direction, or use the importance sampling, sampling from the point light source's direction directly without randomly picking the incoming light direction. Those methods are widely used in industrial rendering and are suitable for different application senarios.

4.2 Results and comparisons for direct illuminance with different sampling methods

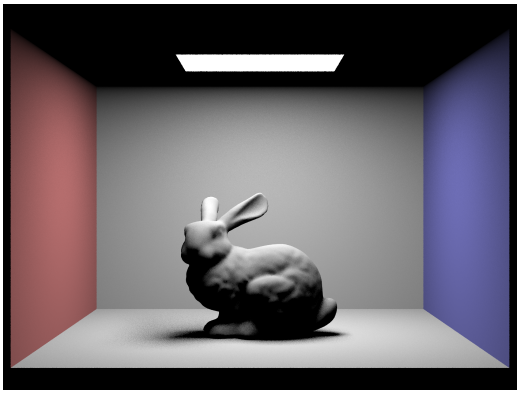
The results for the direct illumination of *CBbunny.dae*, given hemisphere sampling and importance sampling methods, are shown in figure 4. We may observe that the importance sampling method gives much better resutls (with much fewer noises) than that given by the uniform sampling method. This can be explained by the fact that the importance sampling method, instead of randomly sampling from the hemisphere, uses the heuristics of the light source information, making the sampling direction decision more reliable.



(a) Uniform hemisphere sampling: result

```
graphics@Graphics-CUHKSZ:~/Desktop/HW6/rt$ ./pathtracer -t 8 -s 4 -l 8 -H -f CBbunny_hemi.png -r 1920 144
0 dae/sky/CBbunny.dae
[PathTracer] Input scene file: dae/sky/CBbunny.dae
[PathTracer] Rendering using 8 threads
[PathTracer] Collecting primitives... Done! (0.0166 sec)
[PathTracer] Building BVH from 28588 primitives... Done! (0.5050 sec)
[PathTracer] Rendering... 100%! (513.2531s)
[PathTracer] BVH traced 87065432 rays.
[PathTracer] Average speed 0.1696 million rays per second.
[PathTracer] Averaged 13.535794 intersection tests per ray.
[PathTracer] Saving to file: CBbunny_hemi.png... Done!
[PathTracer] Job completed.
```

(b) Uniform hemisphere sampling: runtime information



(c) importance sampling: result

```
graphics@Graphics-CUHKSZ:~/Desktop/HW6/rt$ ./pathtracer -t 8 -s 4 -l 8 -H -f CBbunny_imp.png -r 1920 1440
 dae/sky/CBbunny.dae
[PathTracer] Input scene file: dae/sky/CBbunny.dae
[PathTracer] Rendering using 8 threads
[PathTracer] Collecting primitives... Done! (0.0179 sec)
[PathTracer] Building BVH from 28588 primitives... Done! (0.4991 sec)
[PathTracer] Rendering... 100%! (337.5037s)
[PathTracer] BVH traced 83291388 rays.
[PathTracer] Average speed 0.2468 million rays per second.
[PathTracer] Averaged 11.139476 intersection tests per ray.
[PathTracer] Saving to file: CBbunny_imp.png... Done!
[PathTracer] Job completed.
```

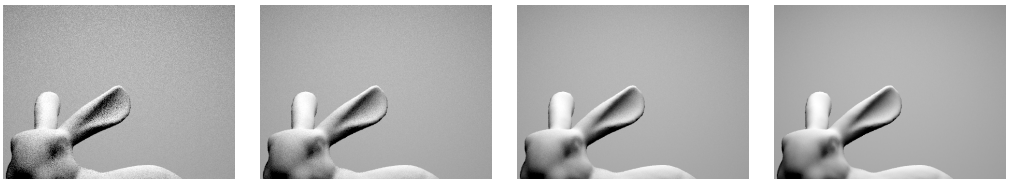
(d) Uniform hemisphere sampling: runtime information

Figure 4: Results and runtime information for different sampling methods

4.3 Noise level comparisons for sampling different number of incoming light rays

The results for sampling with 1, 4, 16, and 64 light rays (adjusted by the -l command line argument), given 1 sample per pixel (the -s flag) using importance sampling method, are shown in figure 5. We may observe that as the number of light samples increase, the noise level decreases

and the number of pixels that are illuminated (having a non-black color, that is, having been traced by some ray from the light source) increases. This is because the increase of the light ray sample rate makes the probability of sampling a ray from a light source (or from an object where the light can be somehow reflected) increases, making it more possible to illuminate the scene.



(a) "-l 1" result (b) "-l 4" result (c) "-l 16" result (d) "-l 64" result

Figure 5: Results for sampling 1, 4, 16, 64 incoming light rays for a particular scene

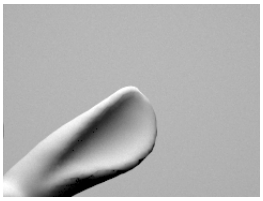
5 Indirect illumination

5.1 Implementations of the at least one bounce illumination function

The indirect illumination indicates illuminating the pixel with rays that are bounced for more than once. Hence, the idea of implementing the indirect illumination can be summarized as a sequence of recursive steps: The outgoing illuminance of the currect intersaction can be equal to the one-bounce illumination of the ray at the current intersaction, plus the Monte-Carlo integration of the at-least-one bounces' illumination provided by the shadow ray and its intersaction with the other object (where the next object's one-bounce illumination is equivalent to the current object's two-bounce illumination, and so on, and hence the recursive steps are valid). Such at-least-one bounced illumation, added with the zero-bounce illumination, consists of the global illumination of the pixel given the ray and its intersaction with the first object.

5.2 Global illumination image result

The result for the global illuminance is shown in figure 6, where we see clearly that the overall illuminance is greatly improved compared to 4 since the result involves the illuminance combining the rays with multiple bounces.

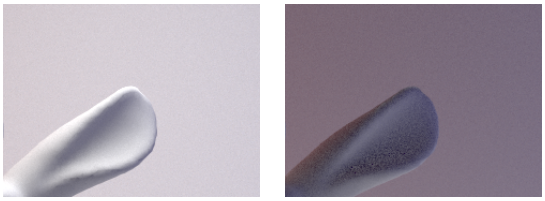


(a) Global illumination result for the rabbit ear

Figure 6: Global illumination result

5.3 Comparisons for direct and indirect illuminations

The results for comparing between the direct illuminations only and indirect illuminations only are shown in figure 7, where we may observe very clearly that the direct illuminations give very smooth and large illuminance, evidently showing the overall shape of the object. The indirect illuminations, on the other hand, shows the edge illuminance (the detailed illuminance with lights going through multiple bounces) better and hence has a generally lower overall illuminance.

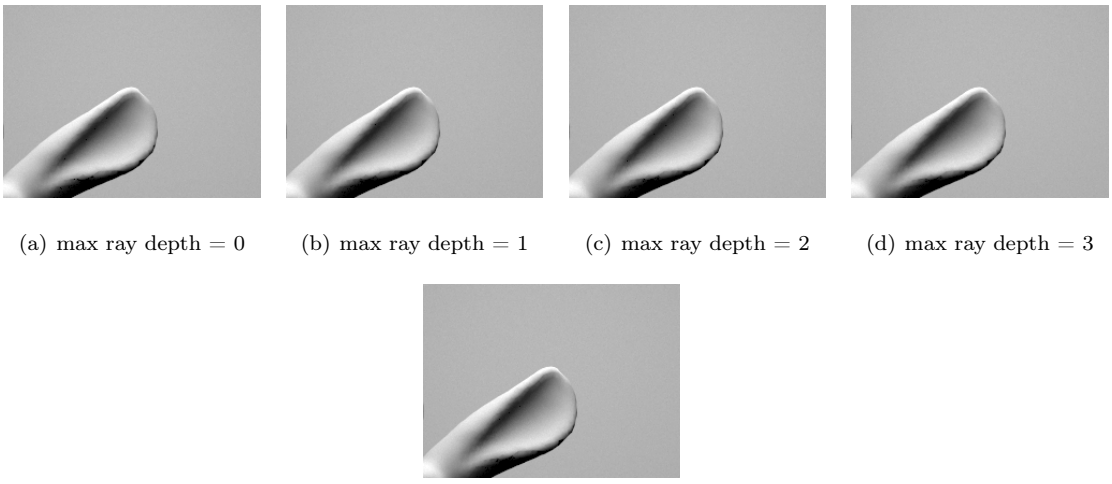


(a) Direct illuminations only (b) Indirect illuminations only

Figure 7: Results for direct and indirect illuminations

5.4 Comparisons for illuminations with different max ray depths

The results for illuminations under different may ray depths are shown in figure 8, where as the ray depth increases, the overall illuminance increases and more details (wrinkles) of the object in the scene are rendered. This shows that the maximum ray depth limits the level of light that one pixel can receive: That is, the higher the depth, the more easily the pixel can capture indirect light rays coming from bouncing many times, hence making the overall illuminance higher as well as the details of the object more visible.



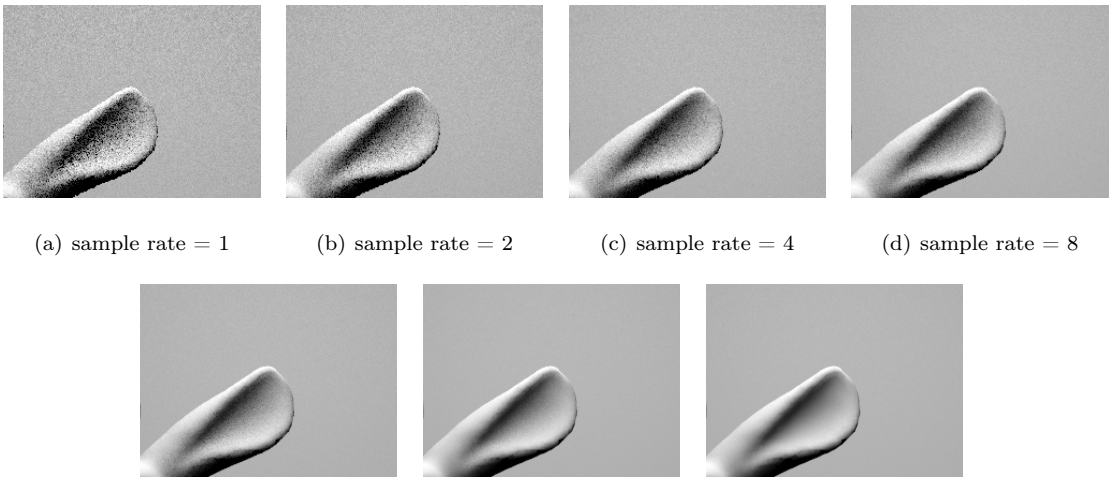
(a) max ray depth = 0 (b) max ray depth = 1 (c) max ray depth = 2 (d) max ray depth = 3

(e) max ray depth = 100

Figure 8: Results with different max ray depths "-m"

5.5 Comparisons for illuminations with different samples-per-pixel with -l 4

The results for different sampling rates are shown in figure 9, where clearly we observe that as the sampling rate increases, the average illumination of the images increases. This is because more light rays are sampled for a given pixel, making it more possible for a pixel to sample from a ray that is "light" (that is, with non-zero illuminance). On the other hand, the running time increases also significantly as the sampling rate increases, indicating the time costs for such stronger illumination. Hence, a tradeoff between the time complexity as well as the illuminance quality should be considered in reality when rendering the global illuminance.



(a) sample rate = 1 (b) sample rate = 2 (c) sample rate = 4 (d) sample rate = 8

(e) sample rate = 16 (f) sample rate = 64 (g) sample rate = 1024

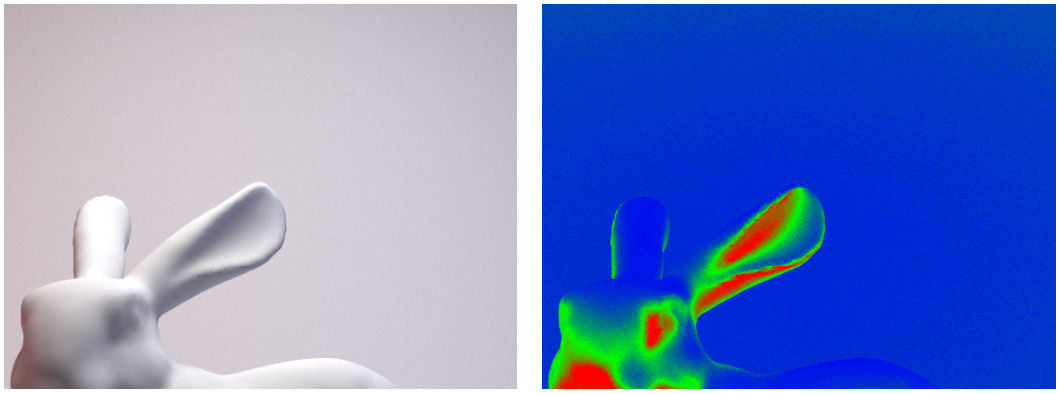
Figure 9: Results with different sample rates "-s"

6 Adaptive sampling method for convergence

6.1 Adaptive sampling implementation

In the adaptive sampling implementation, I sampled rays for a given pixel in batches (with batch size specified by the command line argument "-a xx"). Specifically, I keep adaptively sampling the rays till either the illuminance of the given pixel converges to an appropriate illuminance value (with the confidence of 95 percent) or the maximal number of samples per pixel is reached. The convergence test is hence performed at the end of every batch for better performance and saving the checking time complexity.

6.2 Result for successful adaptive sampling



(a) Adaptive sampling for the scene

(b) Sample rate map

The results for the successful adaptive sampling, given max ray depth>=5, 1 sample per light (-l 1) and 2048 samples per pixel, are shown in figure 6.2. Hence we can observe that the rabbit's eye is heavily sampled (with a high sampling rate) since the region there is relatively dark and it converges more slowly. On the other hand, the surrounding areas don't need too high sampling rates in that they are mostly directly illuminated by the light source (requiring less ray-trace iterations to reach high illuminance) and hence converges faster.