CSC4140 Assignment VI

Computer Graphics

April 23, 2022

Ray Tracing

This assignment is 10%(with 2 extra credit) of the total mark.

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

Student ID: 118010141

Student Name: Jingyu Li

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

Signature: JINGYU LI

Contents

1 Overview

This assignment aims to realize a path-tracing renderer and finish a series of tasks. Ray-tracing technique is vital in computer graphics field since it compensates the weakness of rasterization: the global effect is not satisfied, and hard to deal with the case where light bounces more than one time. Ray-tracing is slow but can obtain a high-quality realistic picture. It usually works off-line instead of real-time rendering. The main mechanism for path-tracing is that we start from camera, focusing on a pixel on image, and cast a ray to a light source (might be an object), where the light bounces a few times, and we calculate the attribute (radiance, color) on the pixel according to the hit-point on the light source. The ray-tracing process really consumes a huge amount of CPU time and patience!

Starting from generate a ray from camera to a specific pixel and implement ray-scene intersection, we then accelerate the ray-scene intersection with bounding box technique and bvh tree. In the third part, I realize bsdf and Zero-bounce Illumination as well as two types of direct illumination: Uniform Hemisphere Sampling and importance sampling. We continue to obtain global illumination image based on direct illumination. Finally, I implement adaptive sampling to apply work on more significant part of the image. This is the road-map of this assignment.

Figure 1: Ray tracing

2 Part 1: Ray Generation and Scene Intersection

2.1 Design and Implementation

Part 1 mainly includes two divisions. The first one requires us to generate ray and pixel samples. The second one asks us to implement ray-primitive intersection.

The generate_ray() function provides the ratio of the pixel in the image - x, y. According to the ratio and camera angles, I firstly obtain the width and height of the image, and then find the coordinates of the pixel. With the object coordinate of the pixel, one can transform it to world space with **c2w** matrix. A ray whose origin is at camera and direction is the world coordinate of the pixel can be constructed now. Remember to set the range of t between nClip and fClip to capture meaningful ray part.

The raytrace_pixel() function gives the coordinate of the pixel as (x, y). we would like to sample rays and calculate the radiance on that pixel. We use a for loop to sample **num_samples** times. Each iteration we use the grid sampler to obtain a specific position on that pixel, estimate the radiance (now we only have normal shading!) and average the values, then write to the sample buffer.

In primitive intersection part, I will go through the more important **intersect** function that assigns values to some variables.

In triangle intersection, it's noticed that about the has_intersection() function, I implemented the original method, which connect the ray equation and the plane to obtain the intersection point, and then express the point as the barycentric coordinates of the triangle to see whether this point lies in the triangle (all three parameters are between zero to one). In intersect() function, I made use of **Moller Trumbore algorithm** directly, which lists as follows.

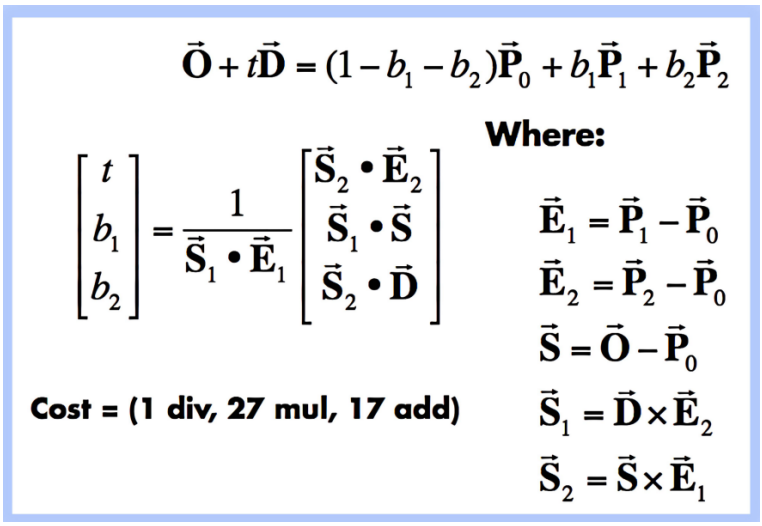

Figure 2: Mollar Trumbore

We can easily obtain the t value of the ray intersection as well as two barycentric coordinates. Similarly, judge whether the intersection point lies in the triangle finishes the task. Remember to set the ray's maximum t value as the current one since we only consider the closest intersection point, also, set the intersection variable. Here shows the code.

```
bool Triangle::intersect(const Ray &r, Intersection *isect) const
{
  // Part 1, Task 3:
  // implement ray-triangle intersection. When an intersection takes
  // place, the Intersection data should be updated accordingly
  // Moller Trumbore algorithm
  Vector3D E1 = p2 - p1;
  Vector3D E2 = p3 - p1;
  Vector3D S = r.o - p1;
  Vector3D S1 = cross(r.d, E2);
  Vector3D S2 = cross(S, E1);

  Vector3D sol = Vector3D(dot(S2, E2), dot(S1, S), dot(S2, r.d)) / dot(S1, E1);
  double t = sol[0];
  if (t >= r.min_t && t <= r.max_t) {
    double b1 = sol[1], b2 = sol[2];
    if (b1 >= 0 && b1 <= 1 && b2 >= 0 && b2 <= 1 && (b1 + b2) <= 1)
    {
      // cout << 1 - b1 - b2 << " " << b1 << " " << b2 << " value " << endl;
      r.max_t = t;
      isect->t = t;
      isect->n = ((1 - b1 - b2) * n1 + b1 * n2 + b2 * n3).unit();
      isect->primitive = this;
      isect->bsdf = this->get_bsdf();
      return true;
    }
  }
  return false;
```

Figure 3: Triangle intersection

For the sphere intersection, the idea is similar. We define a helper function test() to obtain the two intersection values t. By connecting the sphere and the ray and making use of the equation provided by the PPT slides, we can easily solve the quadratic equation and obtain two solutions t1 and t2. In the intersect() function we choose the feasible and smaller t value and assign the intersection attributes.

2.2 Results and Analysis

The followings show images of **.dae** file rendered with normal shading.

The banana image simply represents X coordinate as Red, Y as green, and Z as blue. No blue appears since it's two-dimension. Some colors are mixed. The other three images show a box with empty or an item inside, but the color is used for debugging.

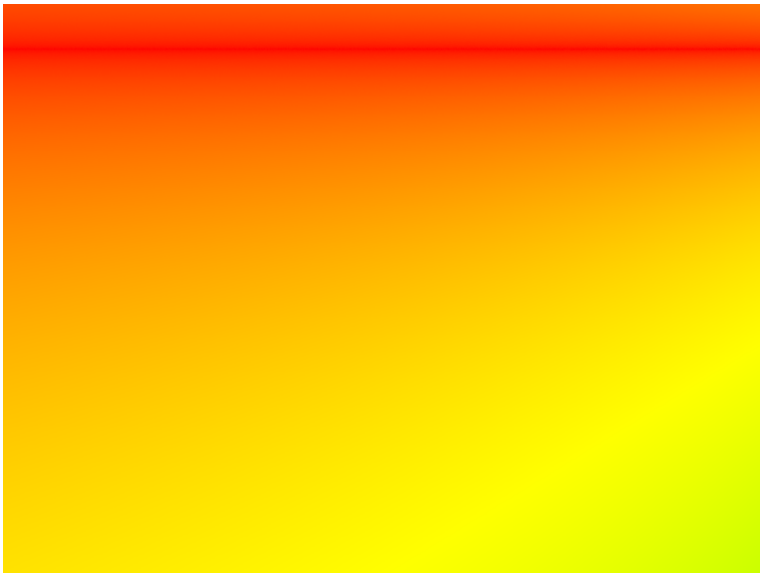

Figure 4: Banana

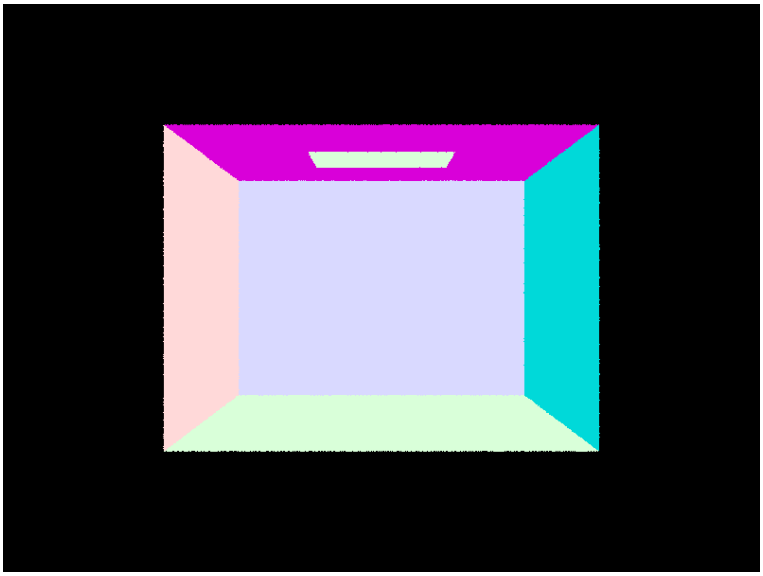

Figure 5: Empty

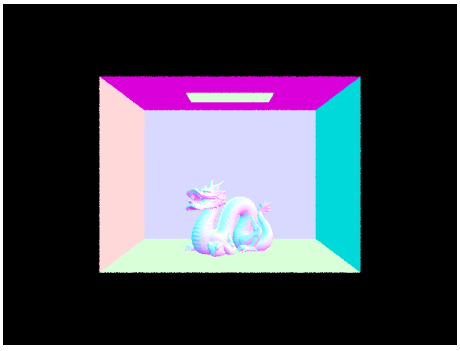

Figure 6: CBdragon part1

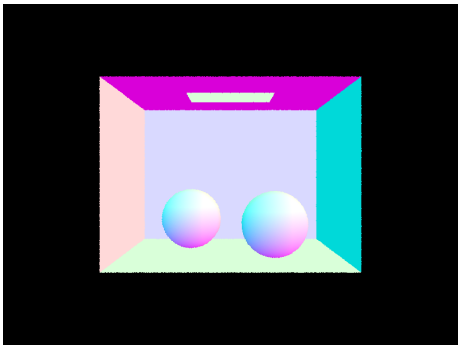

Figure 7: CBsheres part1

3 Part 2: Bounding Volume Hierarchy

3.1 Design and Implementation

Part 2 aims to realize acceleration structure – BVH tree, for intersection judgment. The insight of the idea is that we don't need to judge whether a primitive intersects with a ray by calculating the intersection point one by one. Instead, we can define a bounding box for the primitive. If a ray does not intersect the bounding box, then we terminate the process and claim no intersection happens in this bounding box. BVH tree just divides the whole geometry into different parts for recursive bounding box test. If we find the ray intersects with one node, then we go to its children to perform the intersection test until the end.

In construct_bvh() function, the start and end iterators for the primitive vectors are given, with the maximum leaf size, the maximum number of primitives in a node. I first iterate all the primitives in the vectors, calculating the bounding box for the whole primitive sets as well as the node size. Then I construct a **BVHNode** and treat all primitives as one set. We judge whether this node is leaf node by comparing node size with maximum leaf size. If this is the leaf node, return the node. If not, we need to divide the node further. Here the heuristic I use is **the average of centroids along an axis**. It proves that this heuristic works pretty well. I firstly choose the longest axis among x, y, z by comparing the extent of the bounding box. Then I calculate the average centroids of all primitives. Then I set two vectors, representing left and right child node. Iterate all primitives, if one's centroid is smaller than average centroid, push into left node, otherwise, push to right. Here I check whether either of one is empty, if it is, push one from another side here to make it at least size one. In order not to consume too much memory space, I dump the vector content into the given vector (start -> end) and pass relative part to left node and right node, with the classical recursive call (depth first). Here show the last several steps.

```
for (int i = 0; i < left_primitive->size(); i++) {
  *(start + i) = (*left_primitive)[i];
}
for (int i = 0; i < right_primitive->size(); i++) {
  *(start + left_primitive->size() + i) = (*right_primitive)[i];
}

node->l = construct_bvh(start, start + left_primitive->size(), max_leaf_size);
node->r = construct_bvh(start + left_primitive->size(), end, max_leaf_size);

delete left_primitive;
delete right_primitive;

return node;
```

Figure 8: BVH

In BBox::intersect() function, the ray as well as two reference t values are given to be assigned. Axis aligned ray-bounding box algorithm is utilized. We calculate the t_min and t_max values

of three axes separately, if these values lies in the given range, we pick the maximum among the t_min and minimum among the t_max, to obtain the intersection set. We then set these two values as our new t_min and t_max.

In BVHAccel::intersect() function, we first judge whether the ray intersects the bounding box, if not, we return false immediately. If not, we perform depth-first search, since we need to get the closest intersection point, and iterate left and right child. If we come to a leaf node, go through all the primitives in that node.

3.2 Results and Analysis

The followings show some images with normal shading for a few large .dae files that one can only render with BVH acceleration, or it would be pretty slow!

Figure 9: Cow

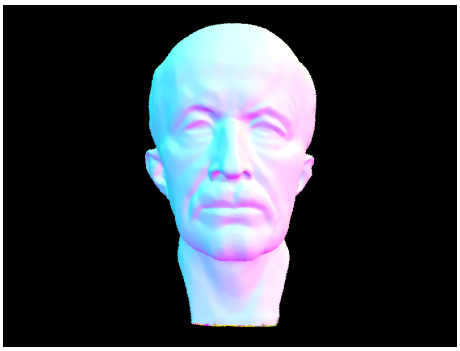

Figure 10: maxPlanck

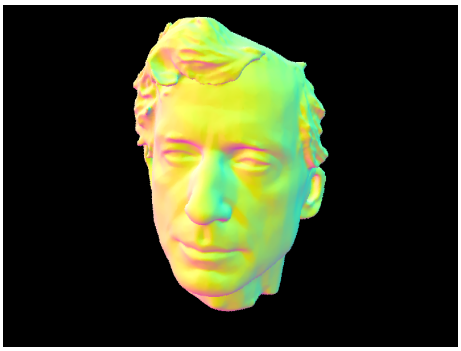

Figure 11: peter

Here also compares the rendering time on the cow scene, a moderately complex geometry with and without BVH acceleration.

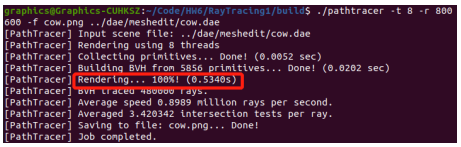

Figure 12: cow without bvh

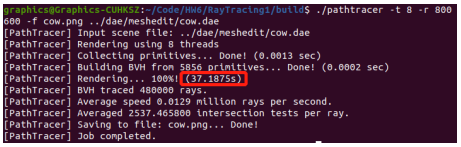

Figure 13: cow with bvh

As seen from above, the rendering time without BVH is 37.1875s and with BVH costs only 0.534s. The difference is 70 multiples!!! So BVH tree structure can accelerate our ray - intersection process greatly. During the test we eliminate many meaning less intersection judgment by testing the bounding box firstly. Since the ray only intersects with a few primitives but not all, we just pick those necessary to check. This is a wise algorithm and really saves quite a lot of rendering time.

4 Part 3: Direct Illumination)

4.1 Design and Implementation

Part 3 requires us to implement direct lighting illumination.

In DiffuseBSDF::f() function, we deal with the case of diffuse reflection. The function is quite simple and We don't even require the parameter wo and wi! We just return the value divided by π. Since the light will reflect to any direction along the hemisphere, so we use the coefficient π to disseminate the radiance.

In zero_bounce_radiance() function, this function returns the radiance resulting from the light without bouncing directly, that is, only the light source. So we just return the intersection point's bsdf's emission field.

The one_bounce_radiance() function returns the radiance for one bounce light, direct illumination. We choose either one of implementation according to a Boolean value and return.

In estimate_direct_lighting_hemisphere() function, the starter code helps us construct the object coordinate system and define hit point and out direction (hit point to camera). We have num_samples in total, which means from the hit point, we can cast these many rays to average the hit point radiance. We should integrate using Monte Carlo estimator.

$$\frac{1}{N}\sum_{j=1}^{N}\frac{f_r(\mathrm{p},\omega_j\rightarrow\omega_r)\,L_i(\mathrm{p},\omega_j)\,\cos\theta_j}{p(\omega_j)}$$

Figure 14: Monte Carlo

We first sample an input direction with the given hemisphere sampler, and define the input ray with origin the hit point and direction the sample one. We should set the minimum t value as a small offset to avoid intersection with hit point. We then intersect the ray with the scene to obtain some intersection information. We find the term in the equation one by one. The f term we defined just now, the L term is the emission of the object intersected, and cos term is obtained by the dot product of input direction and z axis. The p term is the hemisphere arc length 1 π. We average the radiance and return the final result.

In estimate_direct_lighting_importance() has different thought, instead of sampling the ray, the method focuses on the light and samples them directly. For each light source in the scene, we sample the directions between the light and the hit point. If we cast a ray in the feasible direction and no intersection is in the way, we know the light successfully reaches the hit point and can add the radiance into final result. We iterate all the lights in the scene first, and then use sample_L to find a direction and gets its emission. We define a ray in the direction, offset the min and max t value, and perform intersection test. If there is no intersection and the light is above the hit point (z value is larger than 0), we can use Monte Carlo method to integrate the radiance, similarly with previous implementation. The following shows the core part of the importance sampling code.

```
  for (int i = 0; i < num_samples; i++) {
    Vector3D w_in;
    double disToLight, pdf;
    Vector3D L_i = light->sample_L(hit_p, &w_in, &disToLight, &pdf);

    Ray r_in(hit_p, w_in);
    r_in.min_t = EPS_F;
    r_in.max_t = disToLight - EPS_F;
    Intersection isect_sample;
    Vector3D w_in_obj = w2o * w_in;

    if (w_in_obj[2] >= 0 && !bvh->intersect(r_in, &isect_sample)) {
      Vector3D f_win_wout = isect.bsdf->f(w_out, w_in_obj);
      double cos_theta = dot(Vector3D(0.0, 0.0, 1.0), w_in_obj);
      L_sample += f_win_wout * L_i * cos_theta / pdf;
    }
  }
  L_sample /= double(num_samples);
  L_out += L_sample;  // Not divided by light, sum of light
}
return L_out;
```

Figure 15: importance sampling

4.2 Results and Analysis

This section shows some images rendered with both implementations of the direct lighting function - hemisphere and sampling. From the result comparison we can conclude that lighting importance sampling has better effects than uniform hemisphere sampling under the same configuration. It is not only lighter but with more precise details. Hemisphere sampling introduces many noises.

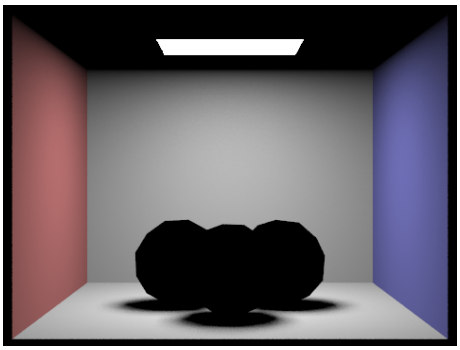

Figure 16: CBbgems 32 16 importance sampling

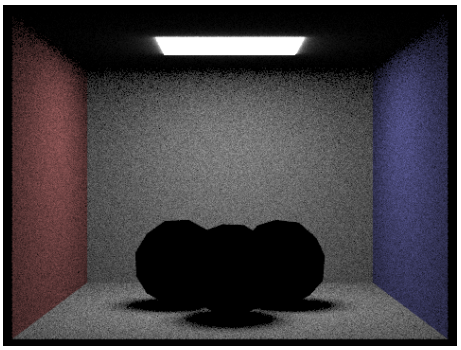

Figure 17: CBbgems H 32 16 hemisphere sampling

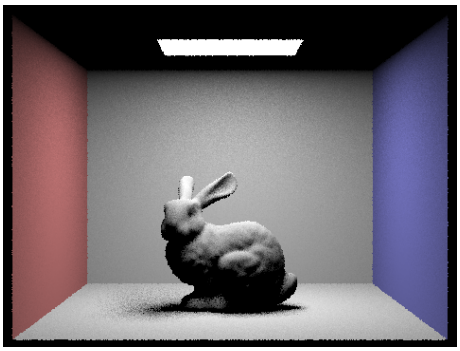

Figure 18: CBbunny 1 16 importance sampling

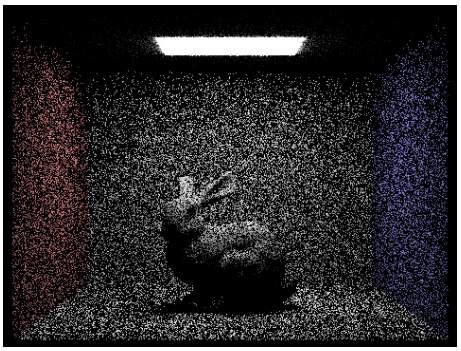

Figure 19: CBbunny H 1 16 hemisphere sampling

Then I focus on one particular scene with at least one area light – I choose the classic bunny box. Rendering with 1, 4, 16, and 64 light rays (the -l flag) and with 1 sample per pixel (the -s flag) using light sampling, not uniform hemisphere sampling, we can compare the noise levels in soft shadows. As the number of light rays increases, the noise level decreases and the quality of images increases. If we only sample a few light rays and not enough, some direction will be no light and appears dark, also the object will not reflects well, introducing high noise levels. When the light rays number is large, we can capture more object detail and obtain better image quality.

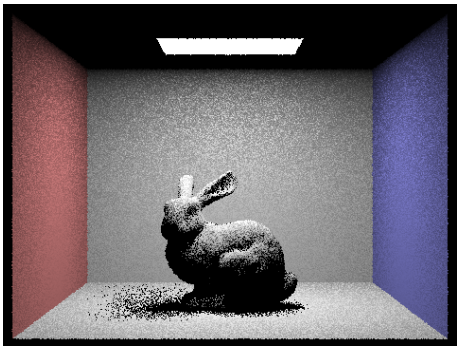

Figure 20: CBbunny 1 1

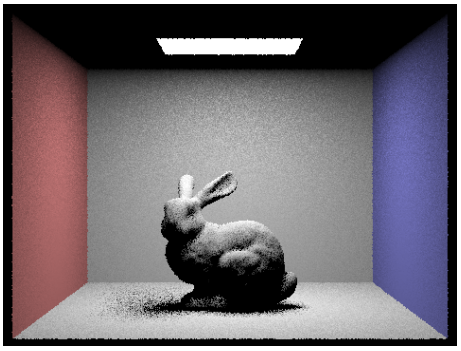

Figure 21: CBbunny 1 4

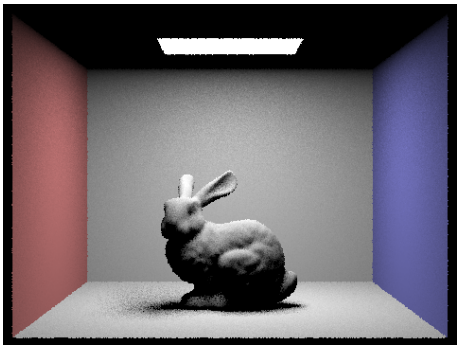

Figure 22: CBbunny 1 16

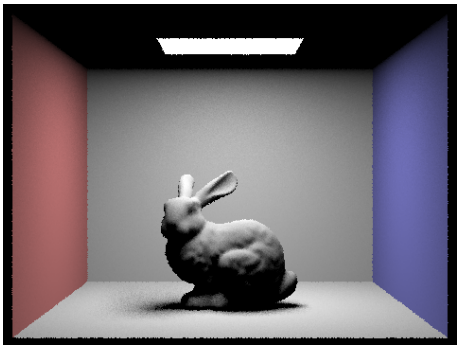

Figure 23: CBbunny 1 64

5 Part 4: Global Illumination

5.1 Design and Implementation

Part 4 requires us to implement indirect lighting function, considering more than one time light bouncing.

In DiffuseBSDF::sample_f() function, it is just an extension of f(). We sample a direction and assign the value to the reference wi, also, we assign probability density function, we return the f(wo, wi) value.

In at_least_one_bounce_radiance(), we implement indirect illumination. We use ray's depth to represent how many times this light can bounce. The maximum value is defined by max_depth. We assign each ray's depth to max_depth at the beginning. If current depth is zero, return zero vector. If not, we then set current radiance as one_bounce_radiance(), invoking the direct illumination. Then we use Russian Roulette to decide whether we should bounce or terminate. The terminate probability is set as **0.3**. Also, if current depth is one, we return the radiance. We then start to come to next level. Firstly sample a direction starting from current hit point, then we define the ray, with a one smaller depth and min_t offset. Then we perform intersection check of the ray and the scene, if we find the intersected primitive, we find the next "light source". Then use Monte Carlo method to integrate the radiance and add up the current radiance. The recursion

happens at the calculate of L term, we should invoke at_least_one_bounce_radiance() again to obtain the radiance at the second hit point (light source). Return the sum radiance.

In est_radiance_global_illumination, now we can obtain the global illumination as (zero_bounce_radiance() + at_least_one_bounce_radiance()), we can then render high-quality lighter image.

5.2 Results and Analysis

The followings will display multiple results regarding part 4.

Here shows some images rendered with global (direct and indirect) illumination. Use 1024 samples per pixel. The effect is pretty good! Beautiful images!

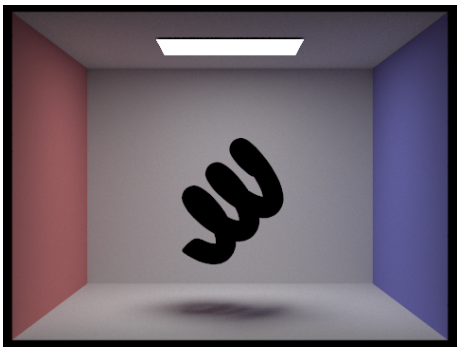

Figure 24: Coil

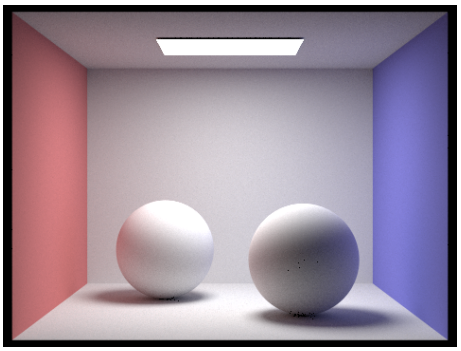

Figure 25: Spheres

Here I pick one scene – the classic bunny image, and compare rendered views first with only direct illumination (the maximum depth is 1), then only indirect illumination (no direct illumination, when the depth is 1, return zero radiance). Use 1024 samples per pixel to ensure high quality. As can be seen below, with only direct illumination, the scene is darker, and indirect illumination provides more light and better quality image, but loses some shadow information.

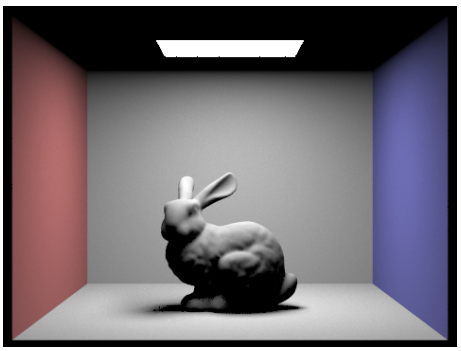

Figure 26: Direct only bunny

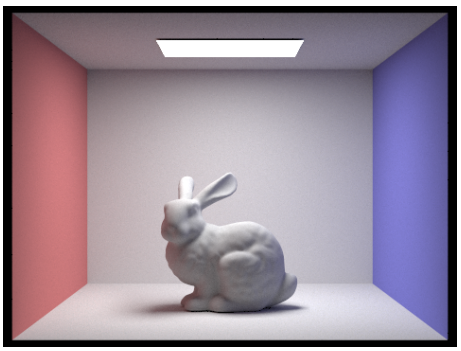

Figure 27: Indirect only bunny

For CBbunny.dae, compare rendered views with max_ray_depth set to 0, 1, 2, 3, and 100 (the -m flag). Use 1024 samples per pixel. As can be seen from the groups of graph, with zero depth, we only have the light coming directly from source. As the light depth becomes larger than 1 and

increases, the image becomes lighter and lighter, showing an illustrious rabbit. The scene seems smoother and smoother. But the detail quality does not change much.

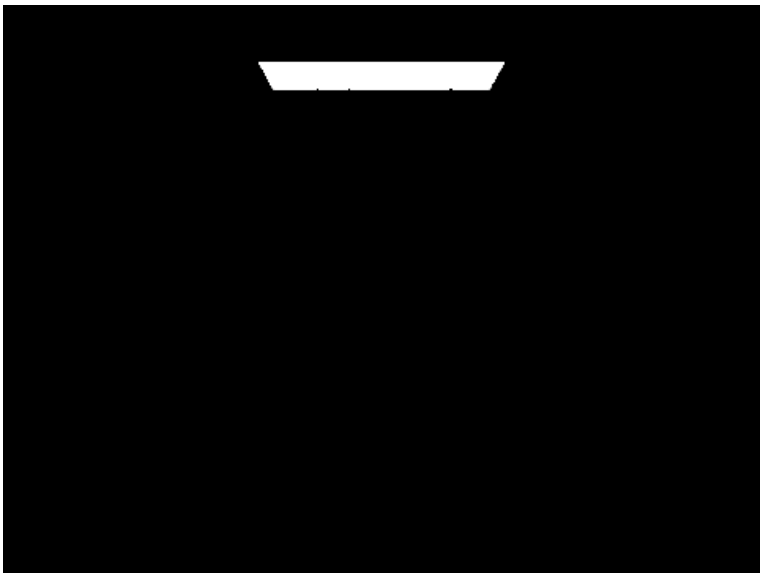

Figure 28: Bunny depth 0

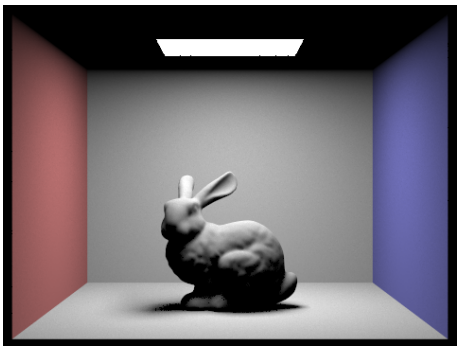

Figure 29: Bunny depth 1

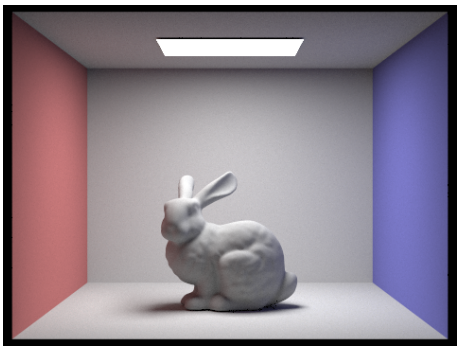

Figure 30: Bunny depth 2

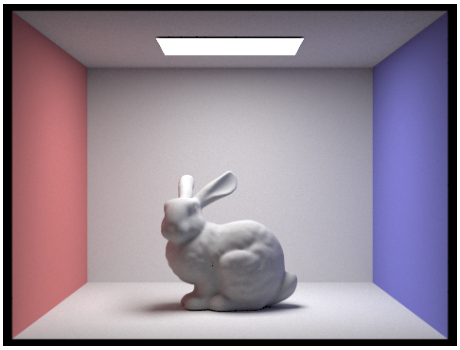

Figure 31: Bunny depth 3

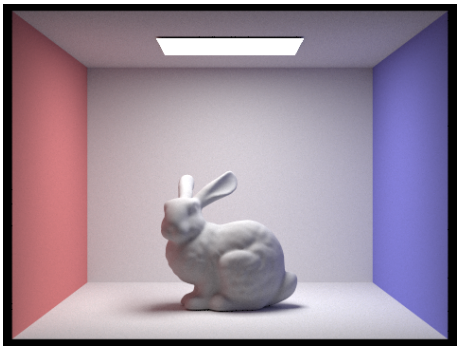

Figure 32: Bunny depth 100

Pick one scene – the special robot and compare rendered views with various sample-per-pixel rates(s flag), including at least 1, 2, 4, 8, 16, 64, and 1024. Use 4 light rays (-l flag). As the sample

rate increases, the quality of the image becomes higher and higher, showing more smooth surface and elaborate details. The noises (black dots on the object) are decreasing.

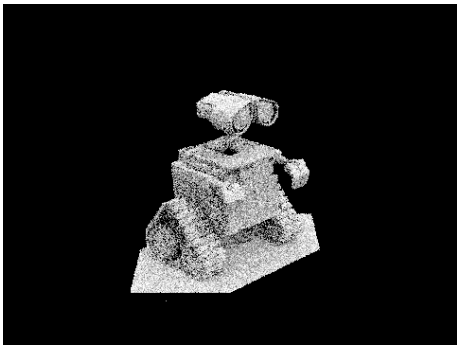

Figure 33: Walle 1

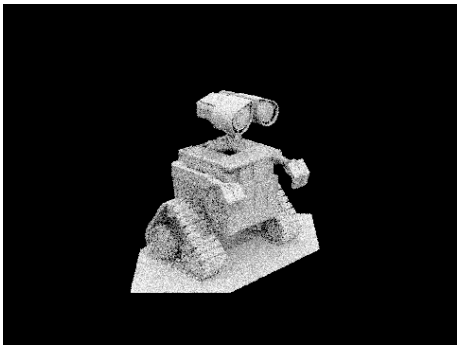

Figure 34: Walle 2

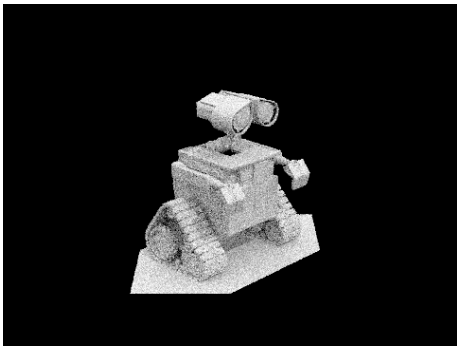

Figure 35: Walle 4

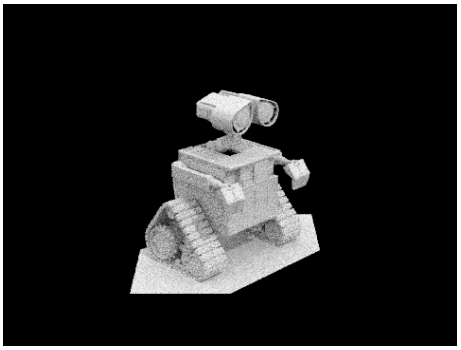

Figure 36: Walle 8

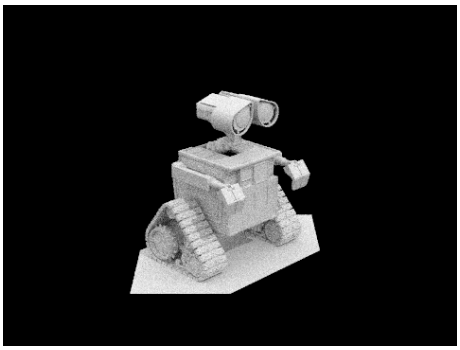

Figure 37: Walle 16

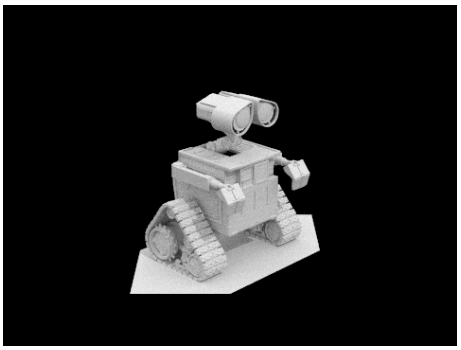

Figure 38: Walle 64

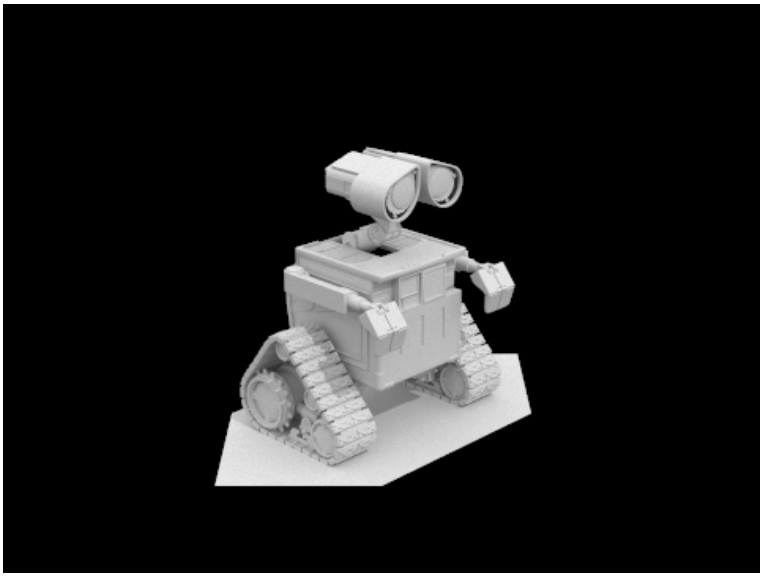

Figure 39: Walle 1024

6 Part 5: Adaptive Sampling

6.1 Design and Implementation

Part 5 requires us to implement adaptive sampling to spend more CPU power on complex part in an image instead of those converge fast and need not sample more. After a long journey, we come back to the function in part 1.

In PathTracer::raytrace_pixel(), we define two terms to record the sum of illuminance and square sum of illuminance in real time. Every time we sample a ray and obtain its radiance, we transform to a scalar, illuminance, and add up the two terms. If the current sample number reaches the multiple of batch number, we start to calculate whether the pixel converges. We get the mean value and standard deviation easily, and we calculate the I term:

$$I = 1.96 \cdot \frac{\sigma}{\sqrt{n}}$$

And judge whether $I \leq maxTolerance * \mu$. If this is true, we stop sampling and average the radiance, assign the value to sample buffer directly and handle next ray. The clever algorithm also accerlates the rendering process.

6.2 Results and Analysis

Here I pick two scenes: bunny and sphere. Both are rendered with 2048 samples per pixel. The corresponding sampling rate image are displayed together. The sample per light equals to 1 and max ray depth is 5.

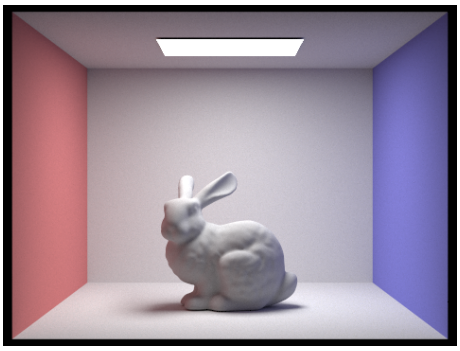

Figure 40: Bunny part 5

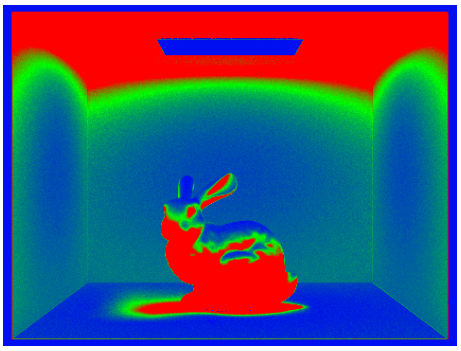

Figure 41: Bunny rate

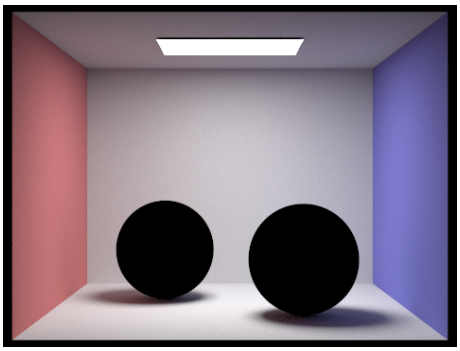

Figure 42: Sphere part 5

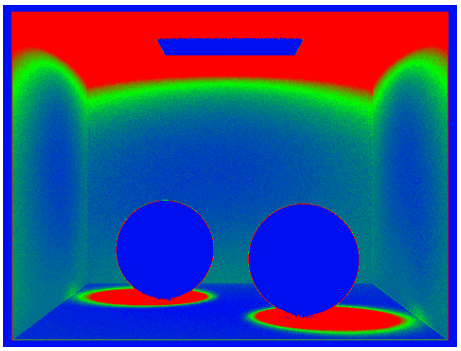

Figure 43: Sphere rate

The results are noise-free, beautiful and comfortable rendering results. The sampling rate image exhibits clearly visible differences in sampling rate over various regions and pixels. The red part represents the high rate while blue is low. The adaptive sampling really works! For the bunny image, the body of the rabbit and the ceiling requires more samples, but the floor and the light source converge pretty fast. For the sphere image, the ball is not diffuse bsdf material, and thus will not reflect light, so the sampling rate is pretty small. But its shadow requires high sampling rate! Wise algorithm.

7 Problems and Solutions

In this section, I will share some problems/bugs/eventful debugging experience when finishing this assignment.

(a) BVH construction - at first I new the vectors in each level to store primitive, space consuming! Should use the same memory space.

(b) BBox intersection: No swapping the t_min and t_max values if their relationship is reverse.

(c) Do not quite understand the Monte Carlo method (for example, what is p?), apply variables to each term solves the problem.

(d) Confuse diffuse bsdf and reflective bsdf, one can reflect light and the other can emit light directly.

(e) Do not quite handle object world coordinate systems at the beginning.

(f) Sample number gets wrong.

(g) Still think Monte Carlo method is a little bit abstract.

(h) No instructional machines for the renders in order to not burn up my own computer for hours!

(i) Floating point comparison is tricky. Should have a small offset.

(j) Forget to transform angle to degree!!

(k) Some typo, forget to add a bracket.

(l) Confusing sign of t1 and t2.

(m) BVH: segmentation fault at the beginning. Should allocate items on heap instead of the stack.

(n) Use a pointer in stack to initialize an object, no actual object.

(o) Confuse two intersections, hit point and light source and camera.

(p) Forget to apply Monte Carlo method in at_least_one_bounce()

8 Execution

(a) Run **sh ./compile_run.sh** to compile and run

(b) Render with fewer samples reduces the time.

(c) Many results are not shown but the graph can be rendered properly.

(d) Use script file to run the experiment.

```
./pathtracer -t 4 -s 1 -l 1 -m 1 -f CBbunny_H_1_1.png -r 480 360 ../dae/sky/CBbunny.dae
./pathtracer -t 4 -s 1 -l 4 -m 1 -f CBbunny_H_1_4.png -r 480 360 ../dae/sky/CBbunny.dae
./pathtracer -t 4 -s 1 -l 16 -m 1 -f CBbunny_H_1_16.png -r 480 360 ../dae/sky/CBbunny.dae
./pathtracer -t 4 -s 1 -l 64 -m 1 -f CBbunny_H_1_64.png -r 480 360 ../dae/sky/CBbunny.dae

# ./pathtracer -t 4 -s 32 -l 16 -m 1 -H -f CBbgems_H_32_16.png -r 480 360 ../dae/sky/CBgems.dae
# ./pathtracer -t 4 -s 32 -l 16 -m 1 -f CBbgems_32_16.png -r 480 360 ../dae/sky/CBgems.dae

./pathtracer -t 4 -s 1024 -l 1 -m 5 -f CBbunny_indirect_only.png -r 480 360 ../dae/sky/CBbunny.dae
./pathtracer -t 4 -s 1024 -l 1 -m 1 -f CBbunny_direct_only.png -r 480 360 ../dae/sky/CBbunny.dae
```

Figure 44: A sample script

(e) Read the code for more details

9 Summary

Here I will summarize what I have learned when writing this assignment.

(a) Great comprehension of ray tracing in computer graphics. For example, ray - primitive intersection, direct/indirect illumination, BVH tree construction, adaptive sampling...

(b) Solve programming problems and have eventful debugging time. Although the workload is pretty pretty heavy!!

(c) Object-Oriented programming paradigm, use class to encapsulate.

(d) Document reading and understanding, much material!

(e) Meaningful time on debugging and coding skill improvement.

(f) Meaningful discussion with peers.

(g) Time Scheduling. I should really start earlier.

(h) Report writing. A high-quality take-away write-up for future review.

(i) By the way, the document and code comments are detailed and clear, Kudos!

That's all.