CSC4140 Assignment 5

Computer Graphics

April 22, 2022

Ray Tracing

This assignment is 10% of the total mark.

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

Student ID: 119010265

Student Name: Shi Wenlan

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

Signature: Shi Wenlan

1 Overview

In this project, I learned and implemented algorithm of camera ray generation, triangle-ray intersection, sphere-ray intersection, Bounding Volume Hierarchy construction and intersection check, diffuse BSDF function, Monte Carlo algorithm, zero-bounce illumination, direct illumination including Uniform Hemisphere Sampling and Importance Sampling, recursive global illumination, and adaptive sampling.

2 Part 1: Ray Generation and Scene Intersection (20 points)

Idea of ray generation: In the function $Camera :: generate\_ray(...)$, I first convert the coordinates of the pixels from the image space to the camera space on the sensor as:

$$x_{camera} = (x_{image} - 0.5) * tan(0.5 * radians(hFov)) * 2$$

$$y_{camera} = (y_{image} - 0.5) * tan(0.5 * radians(vFov)) * 2$$

And represent it using homogeneous coordinate, i.e. $SensorPos = (x_{camera},\ y_{camera},$ -1, 1). Secondly, construct the transform matrix $cameraToWorld$ as:

$$\begin{bmatrix} c2w & pos \\ 0 & 1 \end{bmatrix}$$

Note do not take the transpose of c2w, it will generate a strange output on banana.png (a bug I have ever met). Thirdly, multiply $cameraToWorld$ and $SensorPos$ to compute the world coordinate of the sample point on the sensor $WorldPos$. The ray direction is ($WorldPos$[0] - pos[0], $WorldPos$[1] - pos[1], $WorldPos$[2] - pos[2]).unit() and the ray origin is $WorldPos$[:3] which is the sample point on the sensor.

In the function $PathTracer :: raytrace\_pixel(...)$, I use $gridSampler -> get\_sample()$ to get $num\_samples$ sample points in the given pixel. And use $camera -> generate\_ray(...)$ to generate $num\_samples$ camera rays. Then use $est\_radiance\_global\_illumination(...)$ to get the sample value through the camera rays. After that, average the sample values and store in $finalColor$. Above operations are done in $num\_samples$ iterations, i.e. in each iteration, get one sample point, genrate one camera ray, get one sample value, and $finalColor$ += sample value / $num\_samples$. Finally, update the $finalColor$ to the $sampleBuffer$.

Idea of primitive intersection: In the function $Triangle :: intersection(...)$, I use Möller-Trumbore Algorithm to compute the intersection. The core idea of this algorithm is that the point

of intersection can be represented by both rays $(o + td)$and barycentric coordinates $((1 - b_1 - b_2)p_1 + b_1p_2 + b_2p_3)$, and solve the simultaneous equation $o + td = (1 - b_1 - b_2)p_1 + b_1p_2 + b_2p_3$ can get b, b_1 and b_2. The formula is as following:

$$\vec{\mathbf{O}} + t\vec{\mathbf{D}} = (1 - b_1 - b_2)\vec{\mathbf{P}}_0 + b_1\vec{\mathbf{P}}_1 + b_2\vec{\mathbf{P}}_2$$

$$\begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = \frac{1}{\vec{\mathbf{S}}_1 \bullet \vec{\mathbf{E}}_1} \begin{bmatrix} \vec{\mathbf{S}}_2 \bullet \vec{\mathbf{E}}_2 \\ \vec{\mathbf{S}}_1 \bullet \vec{\mathbf{S}} \\ \vec{\mathbf{S}}_2 \bullet \vec{\mathbf{D}} \end{bmatrix}$$

Cost = (1 div, 27 mul, 17 add)

Where:

$$\vec{\mathbf{E}}_1 = \vec{\mathbf{P}}_1 - \vec{\mathbf{P}}_0$$
$$\vec{\mathbf{E}}_2 = \vec{\mathbf{P}}_2 - \vec{\mathbf{P}}_0$$
$$\vec{\mathbf{S}} = \vec{\mathbf{O}} - \vec{\mathbf{P}}_0$$
$$\vec{\mathbf{S}}_1 = \vec{\mathbf{D}} \times \vec{\mathbf{E}}_2$$
$$\vec{\mathbf{S}}_2 = \vec{\mathbf{S}} \times \vec{\mathbf{E}}_1$$

Figure 1: Möller-Trumbore Algorithm

After solving, judge whether the intersection point is within the triangle and within the visible range according to the barycentric coordinates and t, and update the intersection information accordingly.

In the function $Sphere :: intersection(...)$, I use the following formula to compute the t:

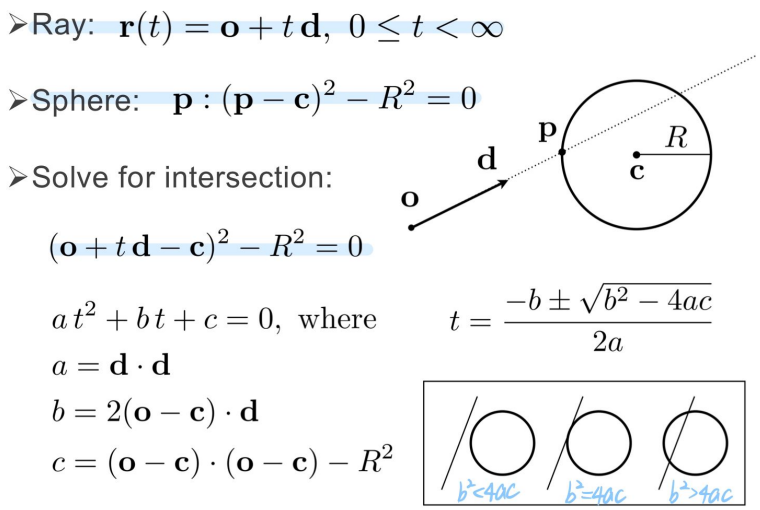

Figure 2: sphere intersection

Note that after computing $b^2 - 4ac$, if its value is negative, return false immediately, otherwise one of the sphere will disappear (a bug I have ever meet). Then after compute t1(closer) and t2(further), judge whether the intersection point within the visible range according to the barycentric coordinates and t, and update the intersection information accordingly.

Images with normal shading for a few small .dae files: The rendered images are as following:

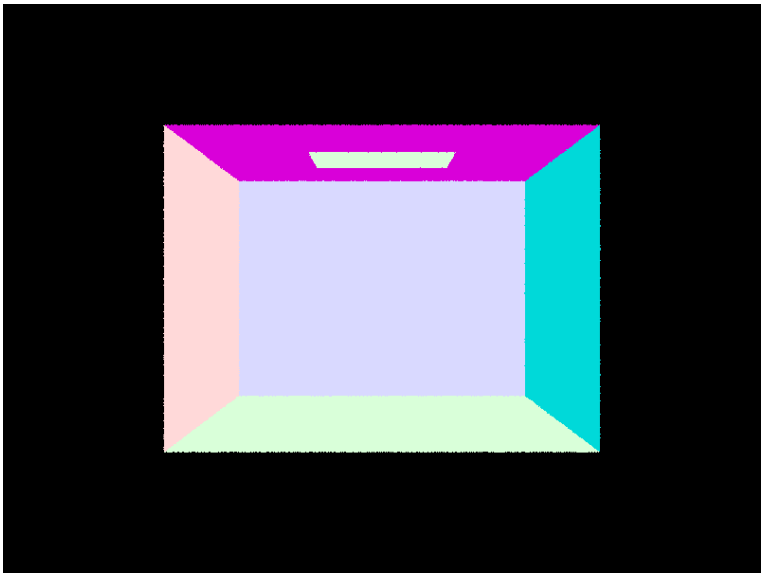

Figure 3: CBempty.png

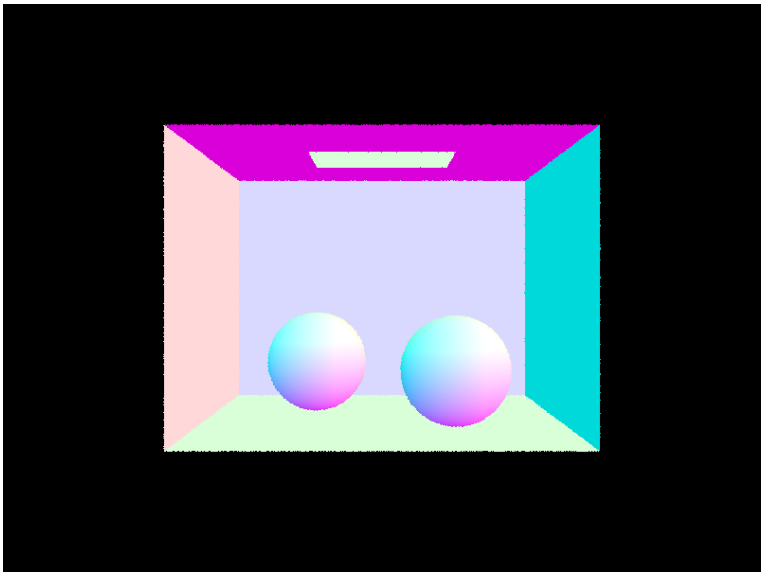

Figure 4: CBspheres.png

Figure 5: cube.png

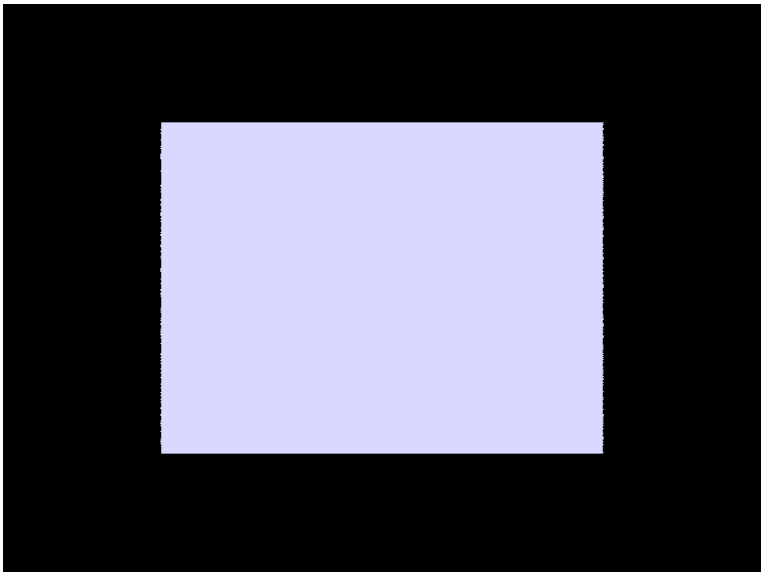

Figure 6: plane.png

3 Part 2: Bounding Volume Hierarchy (20 points)

Idea of BVH construction algorithm In the function $BVHAccel :: construct\_bvh(...)$, I first compute the bounding box *bbox* of given set of primitives and get the number of given primitives *size*. Then construct a $BVHNode$ using *bbox*. If *size* is zero, then return $BVHNode$ immediately. Else if *size* is less than max_leaf_size, store the given primitives to this leaf node and return the node. In other situation, divide the bounding box along the longest extent axis(e.g. if extent.x $>$ extent.y, extent.z, then divide along x axis) at the median primitive (sort according to $primitive.get\_bbox().centroid().x$) and build BVH recursively as assign the left child of

the node $construct\_bvh(smallerHalfPartOfPrimitives)$ and assign the right child of the node $construct\_bvh(LargerHalfPartOfPrimitives)$.

Images with normal shading for a few large .dae files: The rendered images are as following:

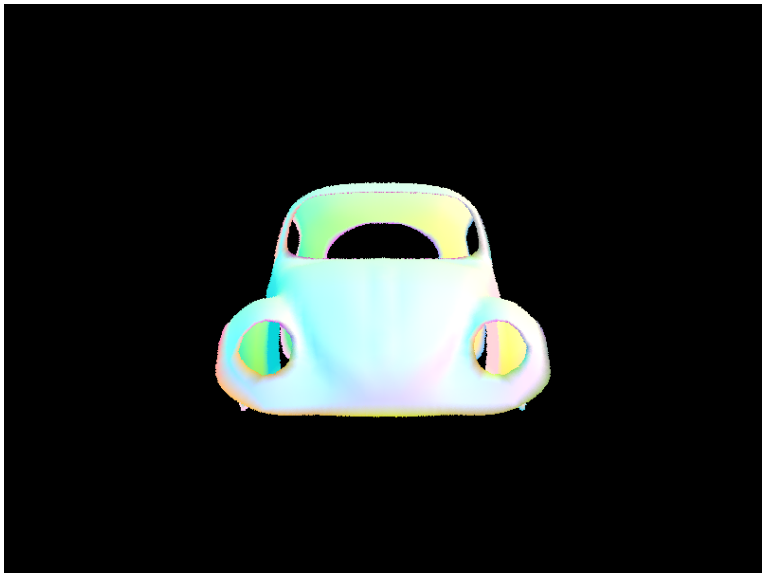

Figure 7: beast.png

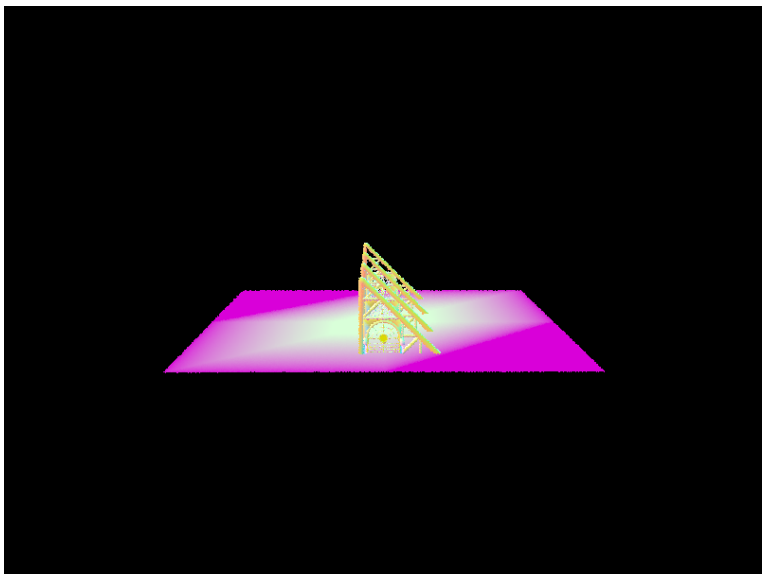

Figure 8: building.png

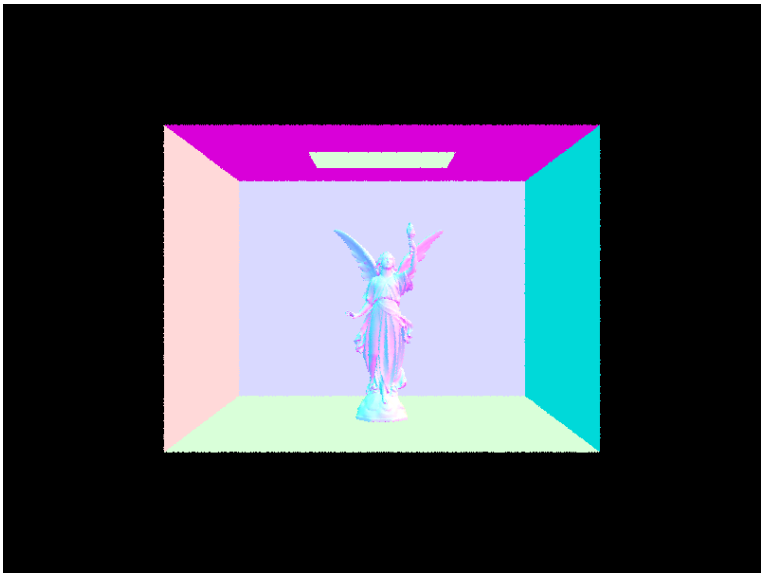

Figure 9: CBlucy.png

Figure 10: maxplanck.png

Compare rendering time: The rendering time before and after using BVH is as following:

Image Name	Before BVH	After BVH
cow	89.5708s	1.4935s
banana	35.6501s	1.8470s
building	very long	1.1648s
beast	very long	1.4682s
beetle	115.1646s	0.9764s
maxplanck	very long	3.5951s
CBlucy	very long	1.6568s

It can be seen from the above table that the acceleration effect of BVH is very obvious. When there are tens of thousands of objects in the scene, before using BVH, each camera ray must be calculated to intersect with tens of thousands of objects, but after using BVH, each camera ray only needs to compute the intersection with less than 10 objects in average.

4 Part 3: Direct Illumination (20 points)

Zero-bounce Illumination: Just return the result of $get\_emission()$ to the closest object that the camera ray directly intersects without ejecting. If the object is a light source, its material should be $EmissionBSDF$, and $get\_emission()$ will return radiance. And if it is an object of other materials, $get\_emission()$ will return 0.

Direct Lighting with Uniform Hemisphere Sampling: I use $num\_samples$ iterations, in each iteration, I do following steps. Firstly, I use $hemisphereSampler- > get\_sample()$ to get a sample direction in the object space. Next, generate a ray ray_w with origin at hit point (note to use EPS_D to avoid numerical precision issues) and direction as sample direction in the world coordinate. Thirdly, try to find the intersection in between ray_w and the scene. If there is an intersection, compute (bsdf function with sampled in-direction and given out-direction) * (the emission of the primitive ray_w intersected, if the primitive is not light source, then this will be 0) * (cos of sample direction in object space) / (pdf = 1/(2PI)) and take the average of all sample directions.

Direct Lighting by Importance Sampling Lights: I traverse all the light sources in the scene, and in each iteration, I do following steps. If it is a point light source, use $light- > sample\_L(...)$ to generate sample in-direction (point from sample point to the light source) in world coordinate, distance between light source and the camera ray origin, pdf and get the radiance of the light source. Next, generate a ray ray_w with origin at hit point (note to use EPS_D to avoid numerical precision issues) and direction as sample direction in the world coordinate, with t_max = distance between light source and the camera ray origin (note to use EPS_D to avoid numerical precision issues). Thirdly, try to find the intersection in between ray_w and the scene. If there is an intersection, then the light is blocked. If there is not any intersection, then compute (bsdf function with sampled in-direction and given out-direction) * (radiance of the light source) * (cos of sample direction in object space) / pdf and add it to the final result. If it is an area light source, nest a loop of ns_area_light iterations, in each iteration, do the steps the same as that of point light source except that the computed result should be averaged by ns_area_light before added to the final result.

Images rendered with both implementations of the direct lighting function: The rendered images are as following (Number of camera rays per pixel = 64, Number of samples per area light = 32):

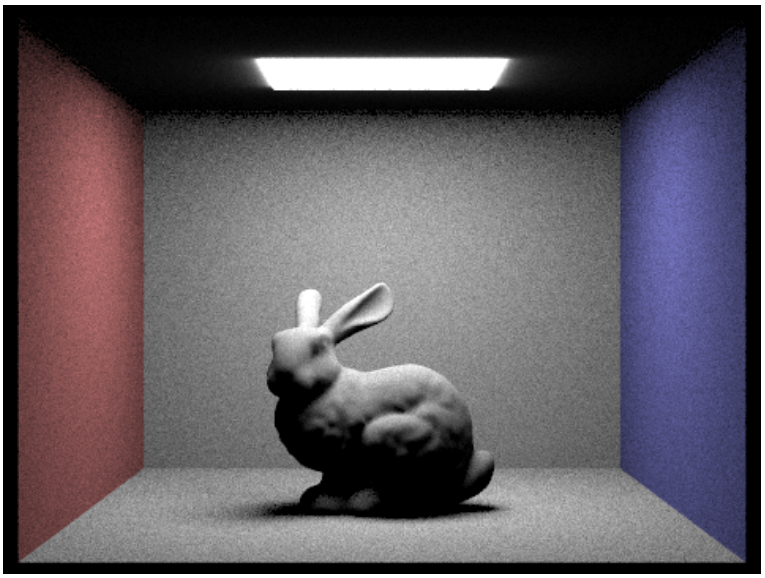

Figure 11: Bunny with Uniform Hemisphere Sampling

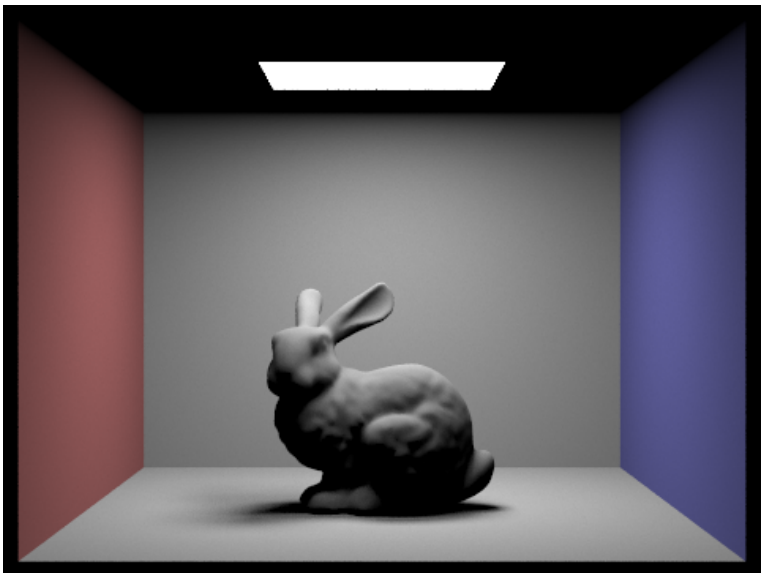

Figure 12: Bunny with Importance Sampling Lights

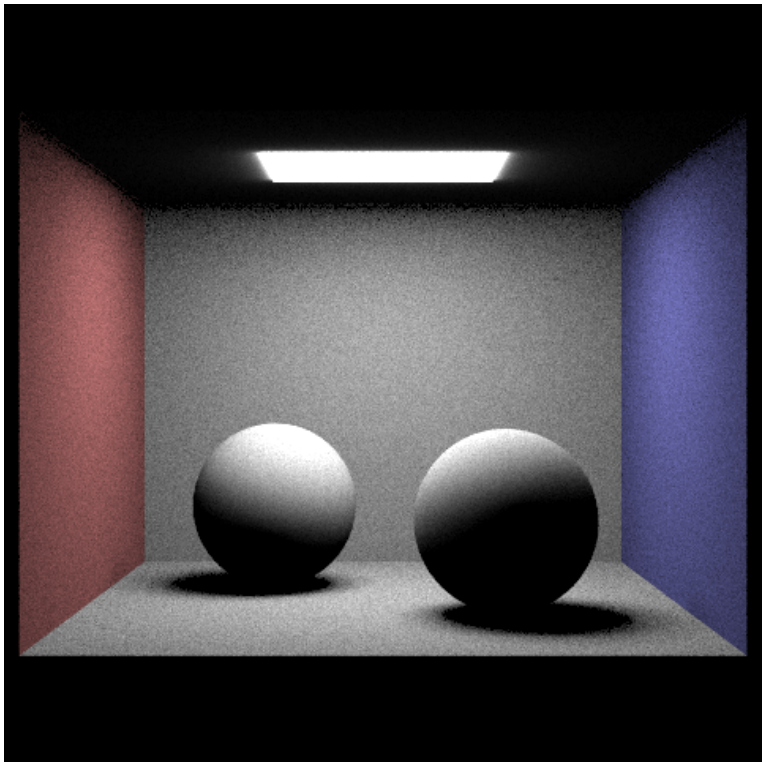

Figure 13: CBspheres lambertian with Uniform Hemisphere Sampling

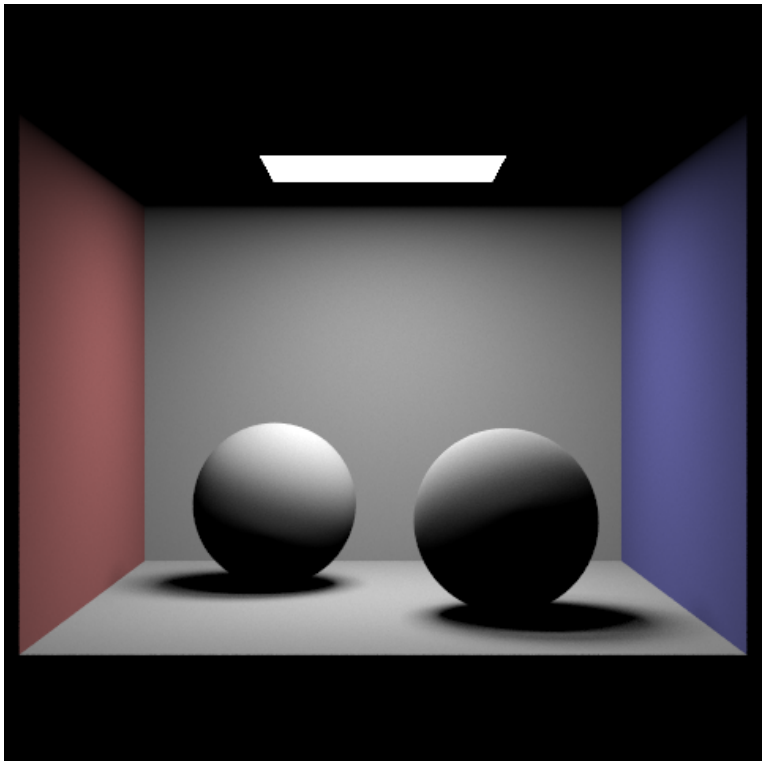

Figure 14: CBspheres lambertian with Importance Sampling Lights

Focus on one particular scene with at least one area light: The rendered images are as following (Number of camera rays per pixel = 1):

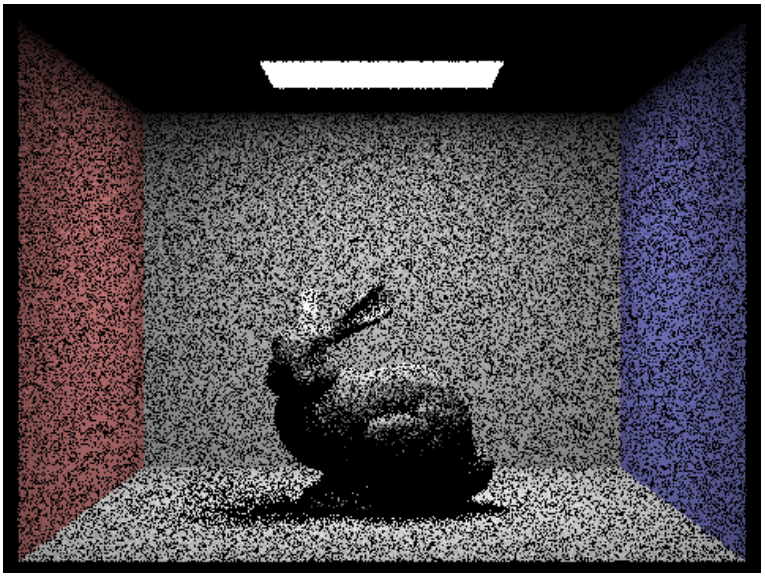

Figure 15: Bunny with 1 light rays

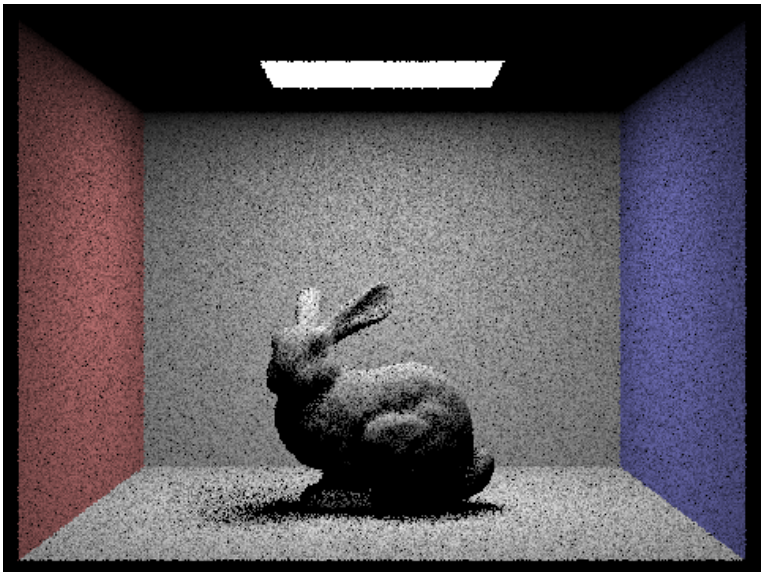

Figure 16: Bunny with 4 light rays

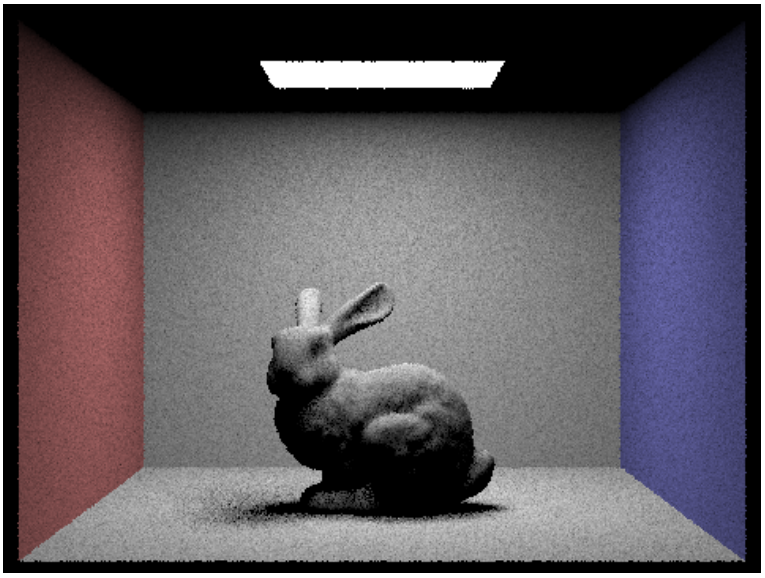

Figure 17: Bunny with 16 light rays

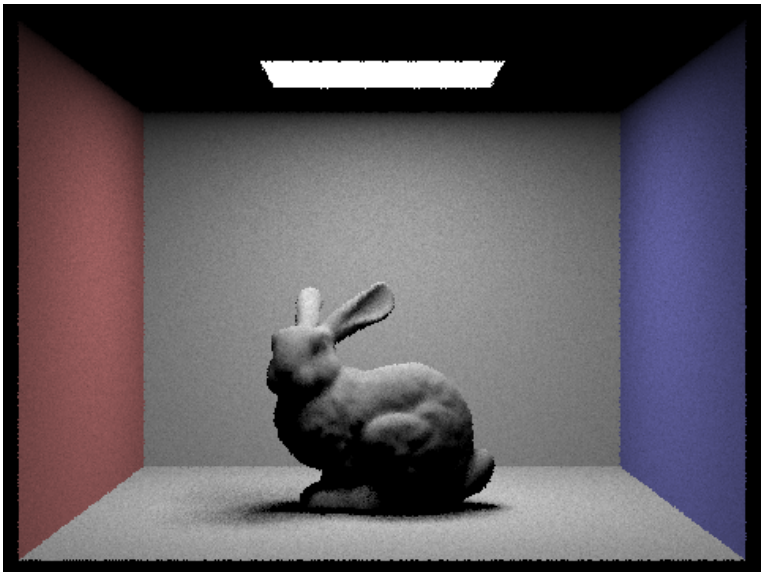

Figure 18: Bunny with 64 light rays

You can see that as the number of samples per area light increase, the noise level in soft shadows decrease.

Comparison between Uniform Hemisphere Sampling and Importance Sampling Lights:

If there is a point P in the scene, we want to calculate the final color of the point P. According to the concept of global illumination, the color of the point P is affected by all the light rays projected on the point P. However, there will be a problem when we use Monte Carlo integral to calculate the sum of projected rays.

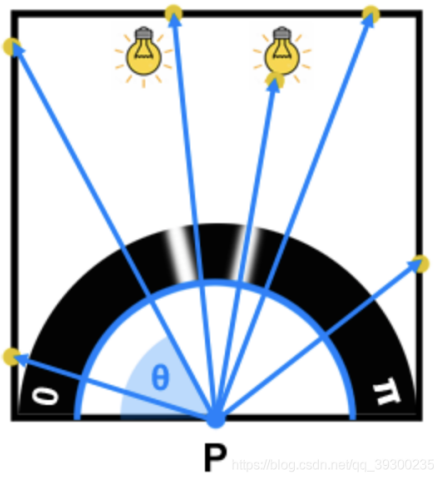

Figure 19: Point P in the Scene

As shown in the above picture, there are two lights in the scene. When there are few Monte Carlo samples, the direction of the lights may not be sampled, so the calculated result will have a huge deviation from the actual situation, especially when the sampling function (f(X)) has a sudden peak. As shown in the following figure.

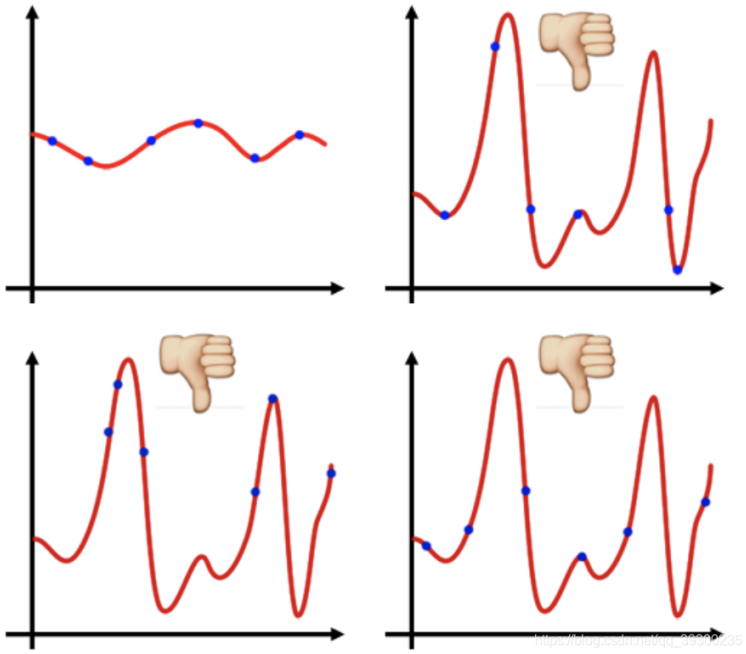

Figure 20: sampling function (f(X))

The pdf Importance Sampling using is proportional to the partial integrand. So the convergence speed of the Importance Sampling method must be faster than that of the Uniform Hemisphere Sampling which use uniform constant pdf. So with the same number of camera rays per pixel and the same number of samples per area light, Importance Sampling has the lower noise level than Uniform Hemisphere Sampling.

5 Part 4: Global Illumination (20 points)

Idea of indirect lighting function: First of all, in the function $est\_radiance\_global\_illumination(...)$, set the return radiance as the sum of $zero\_bounce\_radiance(...)$ and $at\_least\_one\_bounce\_radiance(...)$ if there is an intersection between camera ray and the scene. Then, in the function $at\_least\_one\_bounce\_radiance(...)$, do the following steps. Firstly, add one bounce radiance at the given intersection to the result L_out. Secondly, use $bsdf->sample\_f(...)$ to generate sample in-direction in the object space and pdf. Thirdly, generate a ray ray_w with origin at hit point (note to use EPS_D to avoid numerical precision issues), direction as sample direction in the world coordinate, and depth as 1 + the depth of given ray. Store the recursive depth in the *depth* attribute of *Ray* can help me to end recursion when the recursive depth is equal to max_ray_depth. Fourthly, try to find the intersection *in* between ray_w and the scene. If there is no intersection, then there are no more bounces available, just return L_out. Otherwise check if recursive depth is less than max_ray_depth. If yes, the program recurs the next bounce and add (the returned radiance of recursive function) * (bsdf function with sampled in-direction and given out-direction) * (cos of sample direction in object space) / pdf / (probability of recursion = 0.375) to L_out with a probability of 0.375 (Russ. Roulette). Otherwise, end the recursion and return L_out.

Images rendered with global (direct and indirect) illumination: The rendered images are as following (Number of camera rays per pixel = 1024, Number of samples per area light = 4):

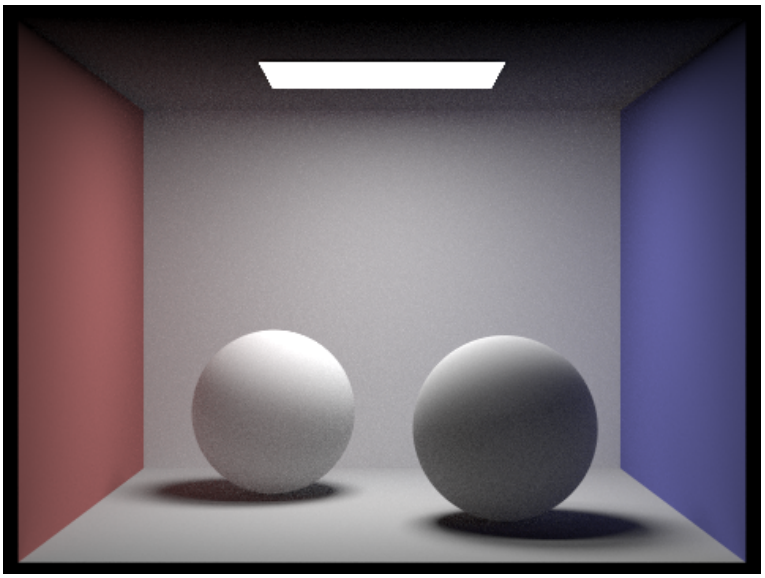

Figure 21: CBspheres lambertian with Maximum ray depth = 5

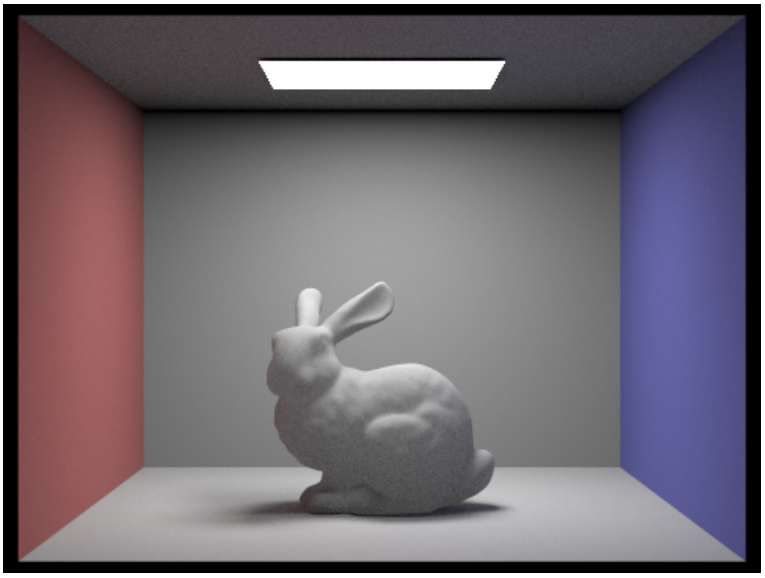

Figure 22: CBbunny with Maximum ray depth = 100

Pick one scene and compare rendered views: The rendered images are as following (dae: CBspheres lambertian, Number of camera rays per pixel = 1024, Number of samples per area light = 4, Maximum ray depth = 5):

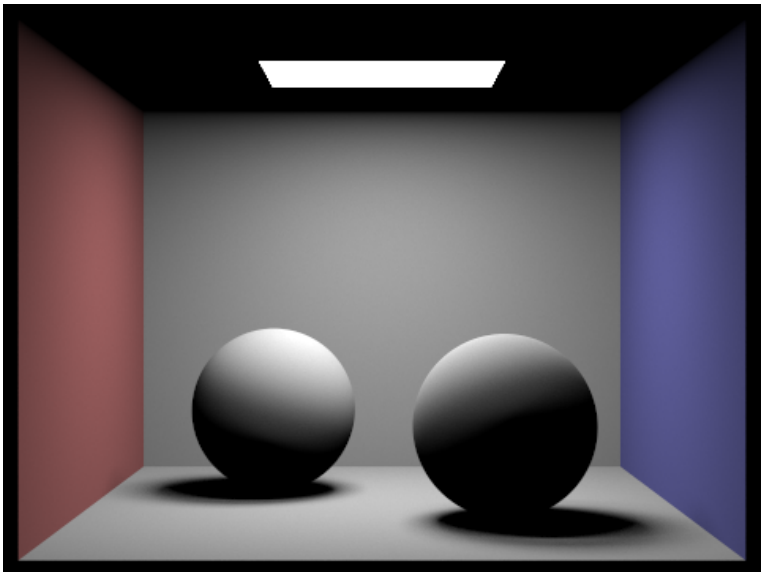

Figure 23: Only direct illumination

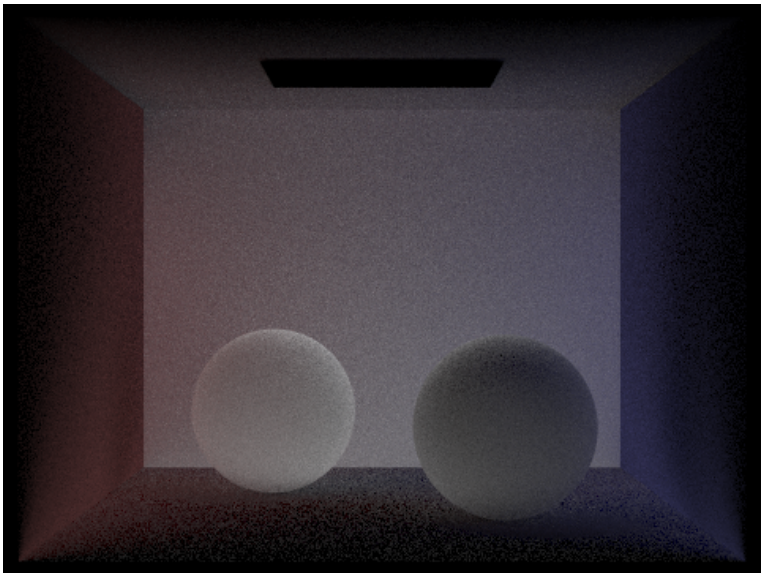

Figure 24: Only indirect illumination

Compare rendered views for CBbunny.dae: The rendered images are as following (Number of camera rays per pixel = 1024, Number of samples per area light = 4):

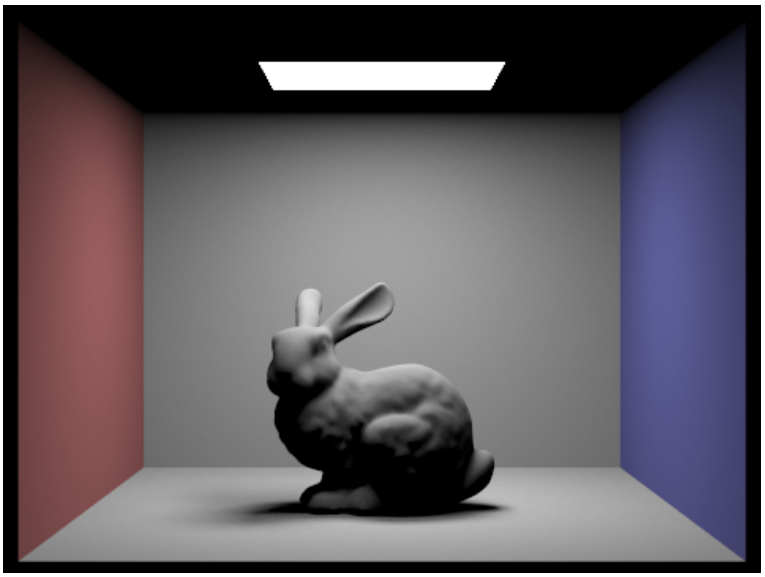

Figure 25: Maximum ray depth = 1

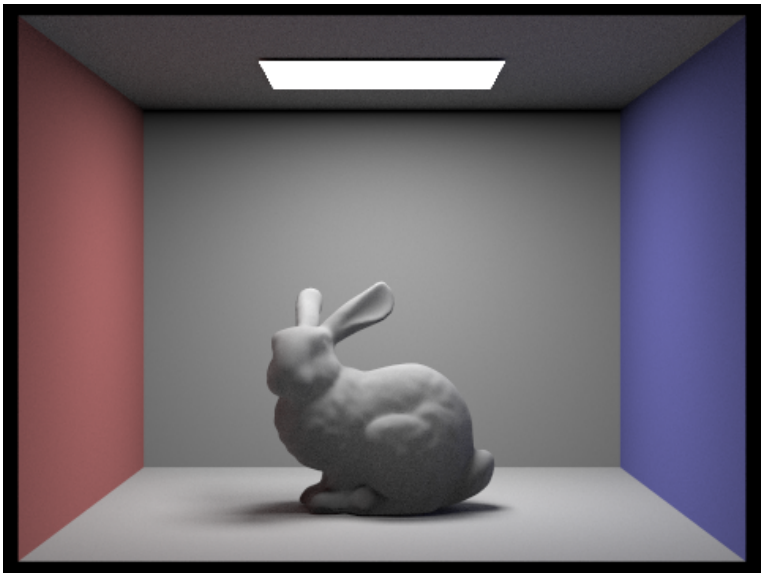

Figure 26: Maximum ray depth = 2

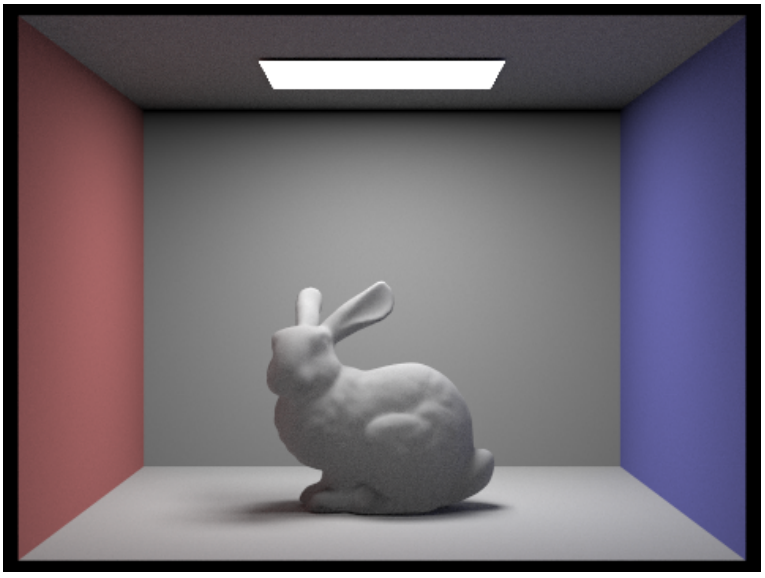

Figure 27: Maximum ray depth = 3

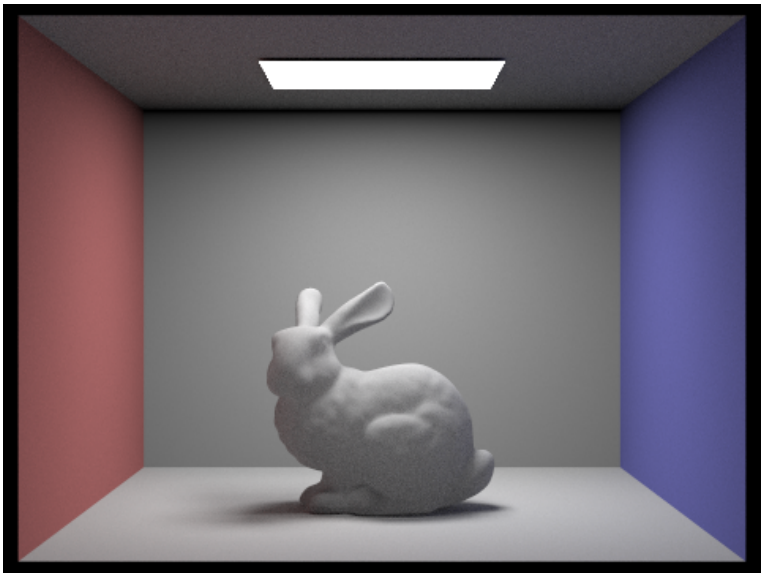

Figure 28: Maximum ray depth = 100

Pick one scene and compare rendered views with various sample-per-pixel rates: The rendered images are as following (dae: CBspheres lambertian, Number of samples per area light = 4, Maximum ray depth = 5):

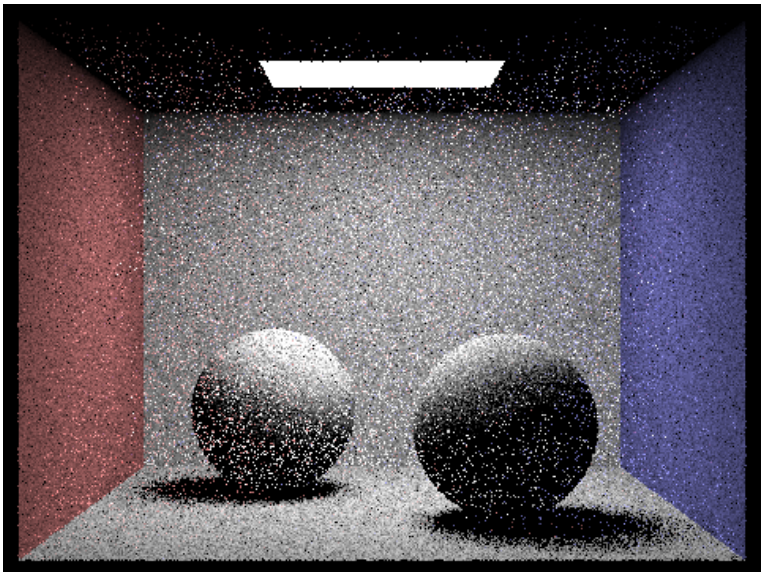

Figure 29: Number of camera rays per pixel = 1

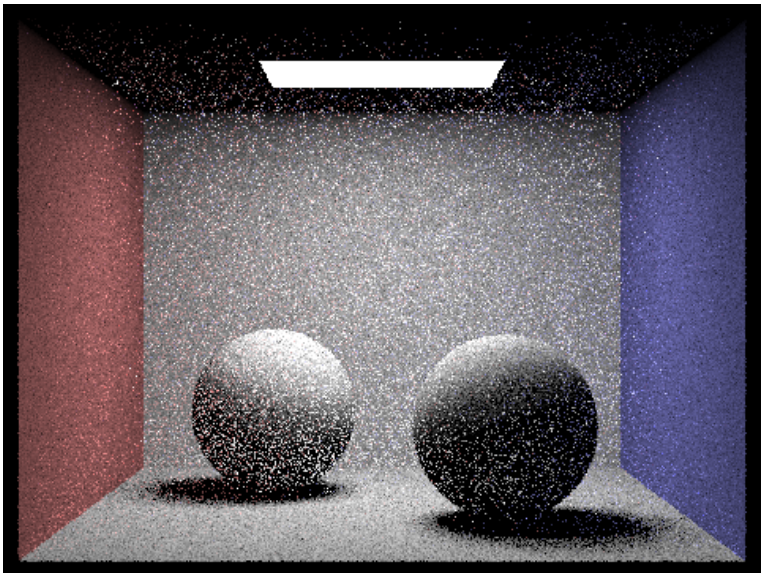

Figure 30: Number of camera rays per pixel = 2

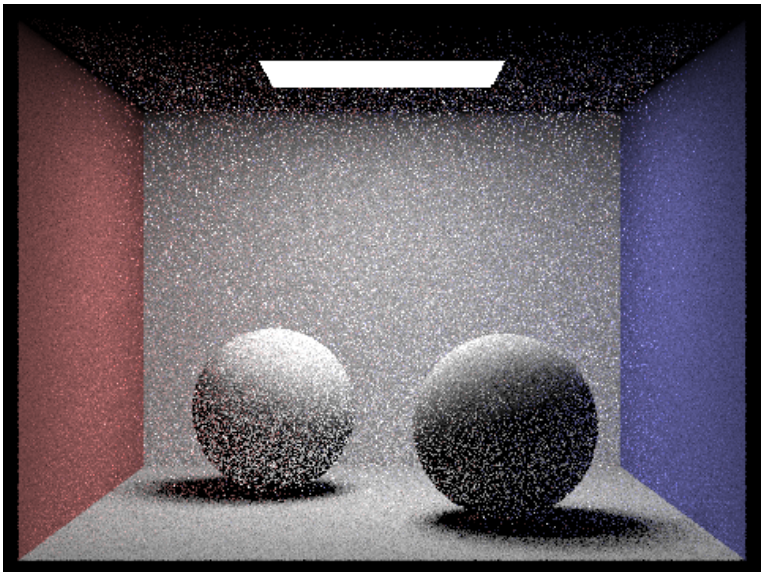

Figure 31: Number of camera rays per pixel = 4

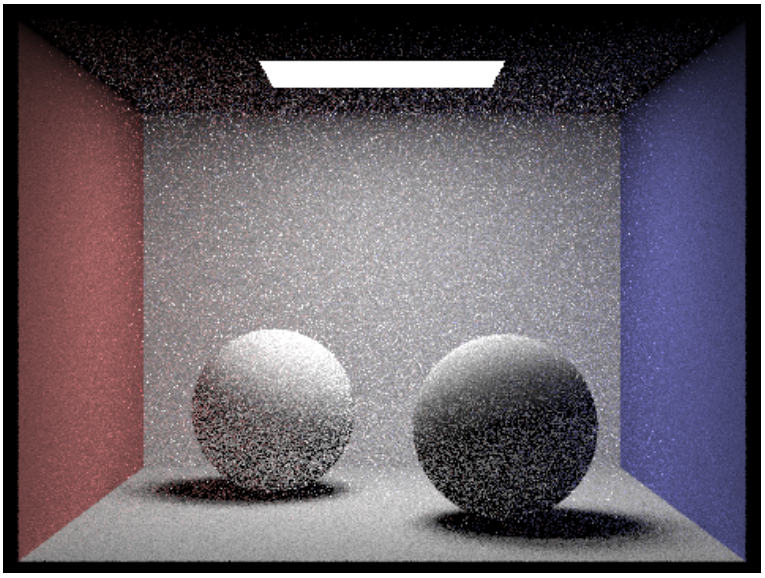

Figure 32: Number of camera rays per pixel = 8

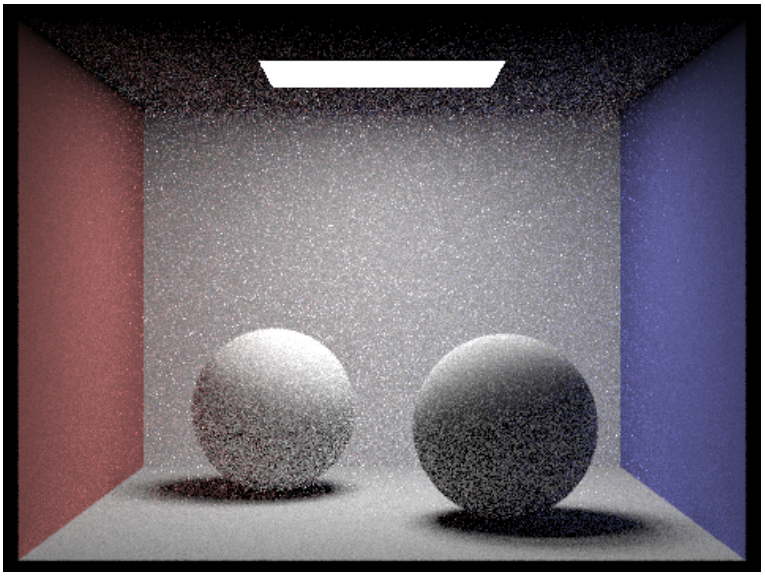

Figure 33: Number of camera rays per pixel = 16

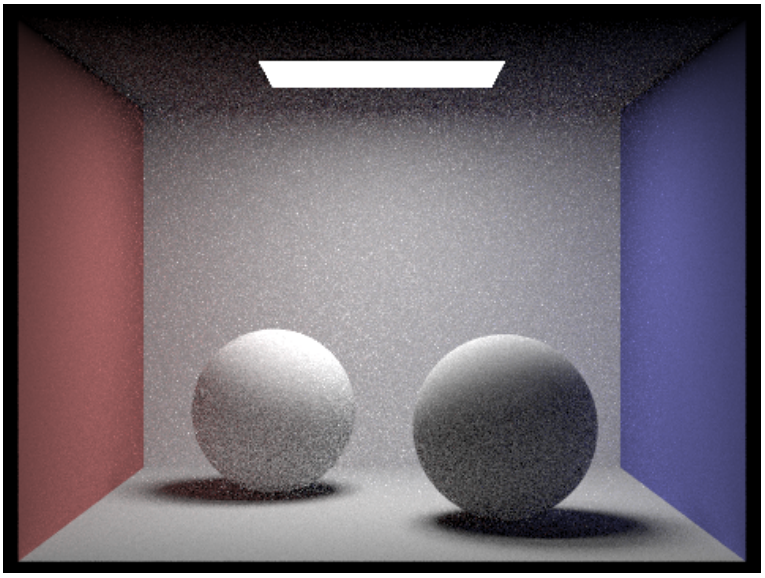

Figure 34: Number of camera rays per pixel = 64

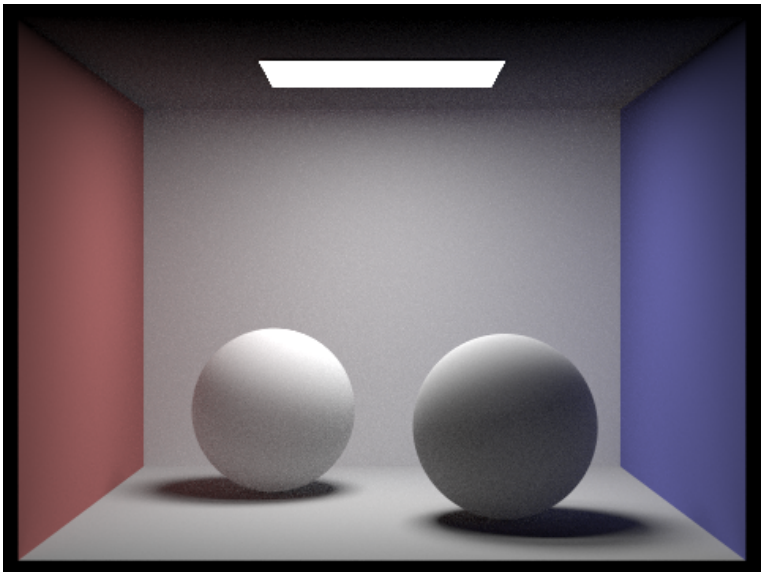

Figure 35: Number of camera rays per pixel = 1024

6 Part 5: Adaptive Sampling (20 points)

Idea of adaptive sampling: First of all, will have at most ns_aa iterations (I may break halfway). In each iteration, I do the following steps. Firstly, I add $num\_samples$ by since I would use $num\_samples$ to record the number of samples has been traced. Secondly, take sample and get the radiance as explained in part 1. Thirdly, use $illum()$ to compute the illuminance of the radiance and stored in xk. Fourthly, add radiance to $finalColor$ so as to calculate the average value later, add xk to $s1$ so as to calculate the mean value later, add xk^2 to $s1$ so as to calculate the variance value later. Fifthly, if $num\_samples$ mod $samplesPerBatch$ equals 0 (this branch

will be entered every $samplesPerBatch$ loops), then compute mean value and variance value using following formula:

$$\mu = \frac{s_1}{n}$$

$$\sigma^2 = \frac{1}{n-1} \cdot (s_2 - \frac{s_1^2}{n})$$

Figure 36: formula of mean and variance

And compute I which is used to measure the the pixel's convergence using following formula:

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

Figure 37: formula of I

If I is less than or equal to $maxTolerance$ * variance, then I regard the value of the pixel has converged, and I can break the loop immediately. Sixthly, outside the loop, divide $finalColor$ with $num\_samples$ to take the average, and update $finalColor$ to $sampleBuffer$, $num\_samples$ to $sampleCountBuffer$.

Pick one scene and render it: The rendered images are as following (dae: CBbunny, Number of samples per area light = 1, Maximum ray depth = 5, Samples per batch = 64, tolerance for adaptive sampling = 0.05, Number of camera rays per pixel = 2048):

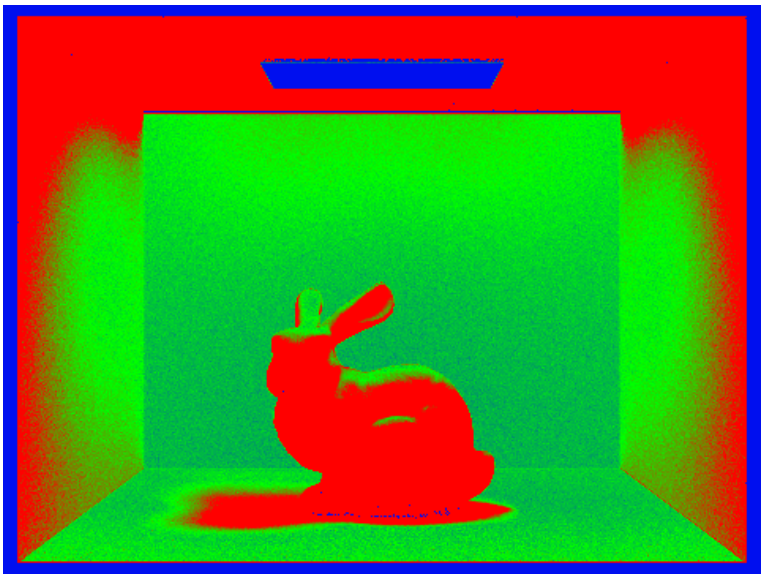

Figure 38: sample rate image

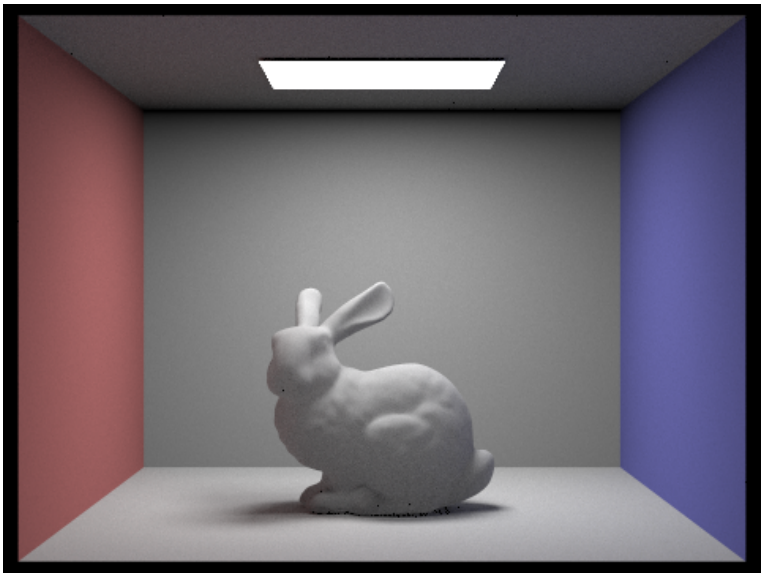

Figure 39: noise-free rendered result

7 Debug Partner: 118010092 He Jingjing

Hardest bug I have met: When I implement part 3 task 3, I met a bug that the wall would be half-triangle light and half-triangle dark, and the floor also has half-triangle corrugation, as figure show below:

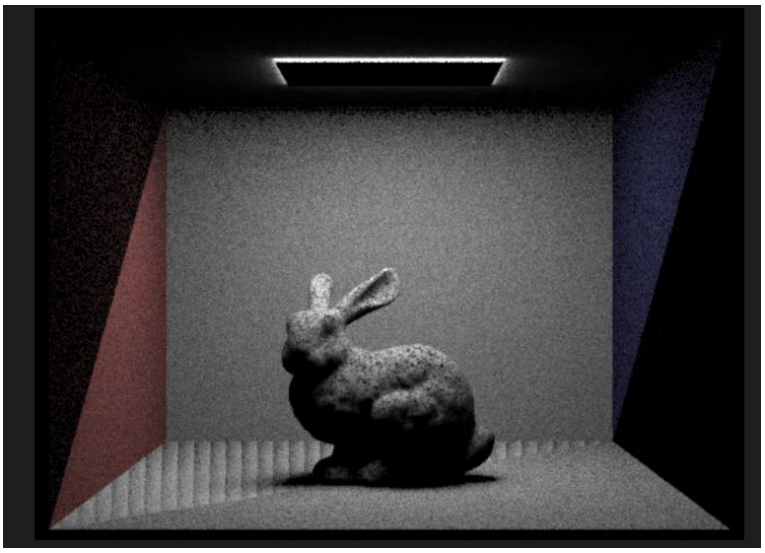

Figure 40: bugBunny

How to fix it: She has complete part 3 at that time. I send my code to her, I sent her my code, and she replaced my files into her framework and checked which file cause the bug one by one. I detailed my code ideas to her and discussed where the problem might be. Finally, it was found that the bug came from part 1.1 in *camera.cpp*, the origin of the ray generated was set to

the origin of the camera instead of the sampling point on the sensor, which would cause errors in the calculation of subsequent checks for intersection, and in part 3.3 of *pacetracer.cpp*, the ray generation does not using EPS_D to offset the origin, results in computational precision errors that are not avoided.