CSC4140 Assignment VI

Computer Graphics

April 21, 2022

Ray Tracing

This assignment is 10% of the total mark.

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

Student ID: 118010335

Student Name: Wei WU

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

Signature:

1 Overview

This assignment is to implement ray tracing for global illumination on diffuse materials. In Part 1, we generate rays from the camera. We then transform the rays from the camera space to the world space using the c2w rotation matrices. We use mathematical methods to calculate ray-triangle and ray-sphere intersections. To visualize the results, we utilize interpolated normals to color the pixels. In Part 2, we accelerate the ray-primitive intersection computation using bounding volume hierarchies. This tree-like data structure allows fast look-up for intersections. In Part 3, we implement direct illumination for Lambertian diffuse materials. In terms of sampling, we implement both direct hemisphere sampling and importance sampling methods. In Part 4, we implement global illumination with path tracing. We bounce each camera ray for a number of times to collect irradiance from non-emissive materials. Global illumination brings features such as indirect illumination, color bleeding, and soft shadows. In Part 5, we improve the sampling efficiency by implementing adaptive sampling. In the scene, some pixels converge faster while others need more iterations. Adaptive sampling stops further sampling for nearly converged pixels, effectively reducing rendering time. To conclude, we implement a render that can produces photo-realistic images with global illumination in reasonable time.

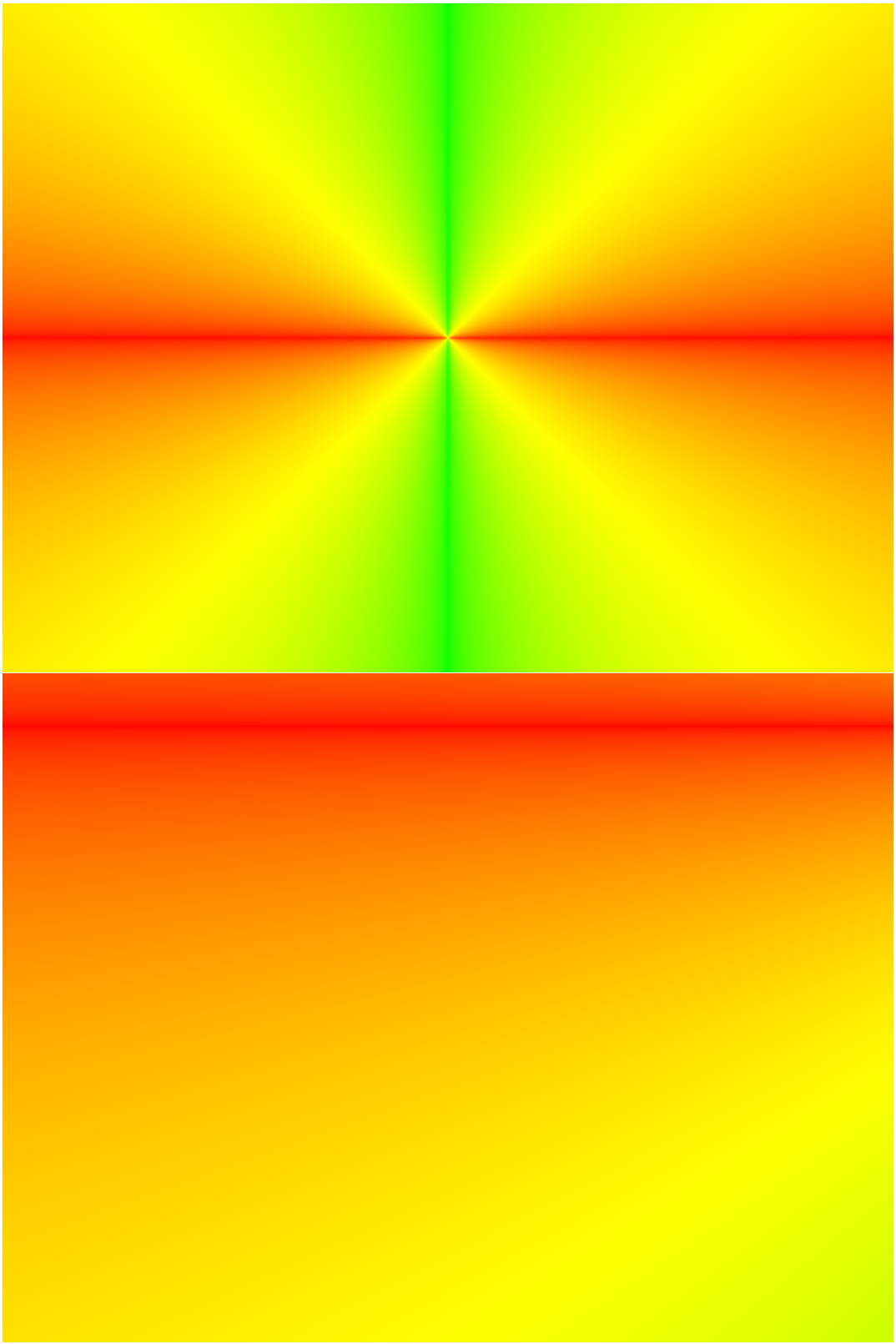
2 Part 1: Ray Generation and Scene Intersection

Within each pixel, we use a grid sampler to generate a certain number of rays in the camera space. The rays originate at the camera and point toward the sample point on the projection plane. Then, we use c2w rotation matrix to transform the ray into the world space. We solve ray-triangle intersection and ray-sphere intersection.

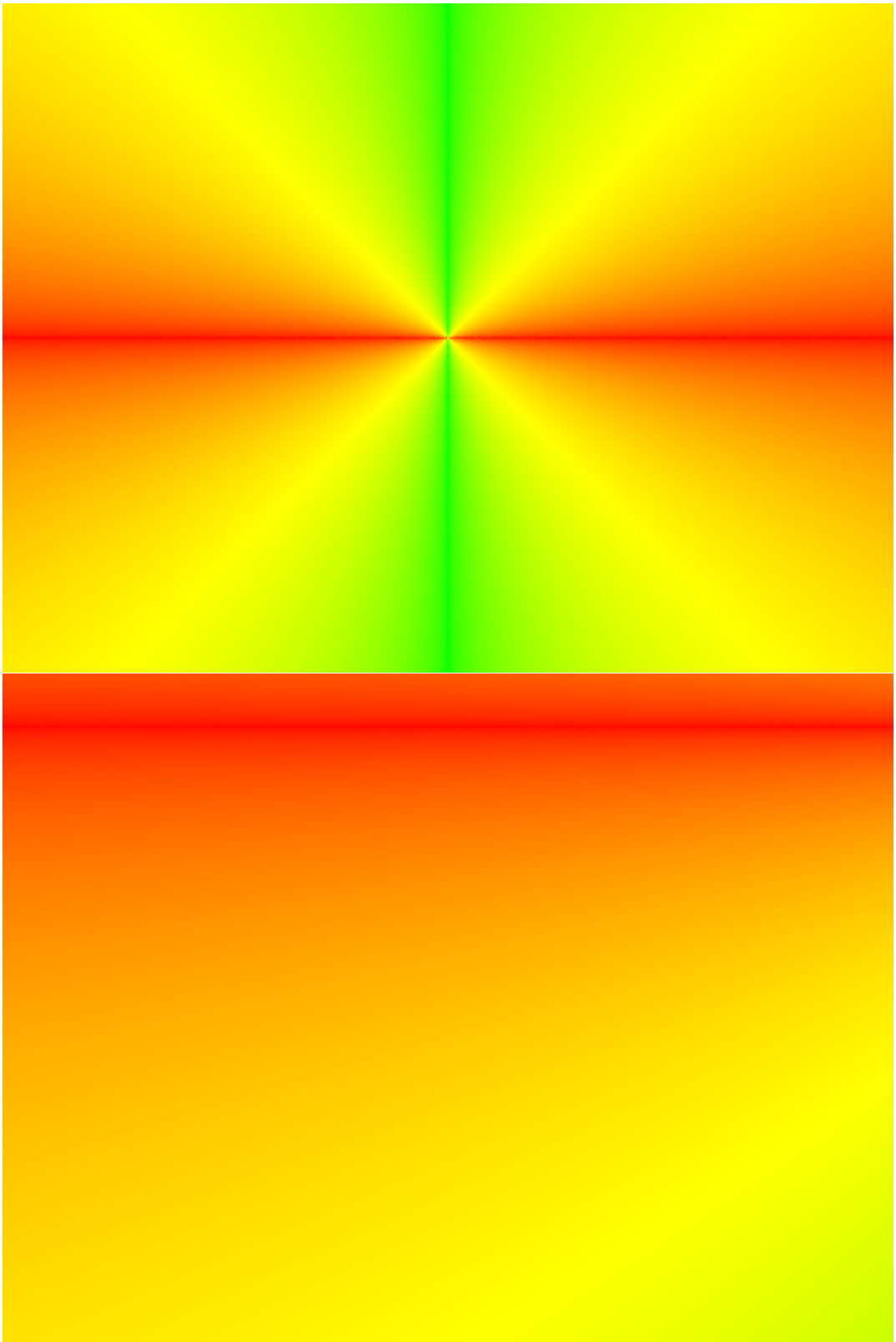
We implement the Möller Trumbore Algorithm to solve ray-triangle intersection efficiently, which calculates the barycentric coordinates for the point of intersection. To be specific, we use Cramer's rule to solve the barycentric coordinates of the intersection using vertex coordinates as well as ray direction. Then, we check if the barycentric coordinates is valid, i.e., the point is inside the triangle. Also, for the ray, we check if t is between the minimum t and maximum t. If the intersection is valid, we update maximum t as the intersection t to eliminate unnecessary computation in the subsequent scene intersection test for this ray. In the meanwhile, we update isect information according to the intersection point.

2.1 Task 2

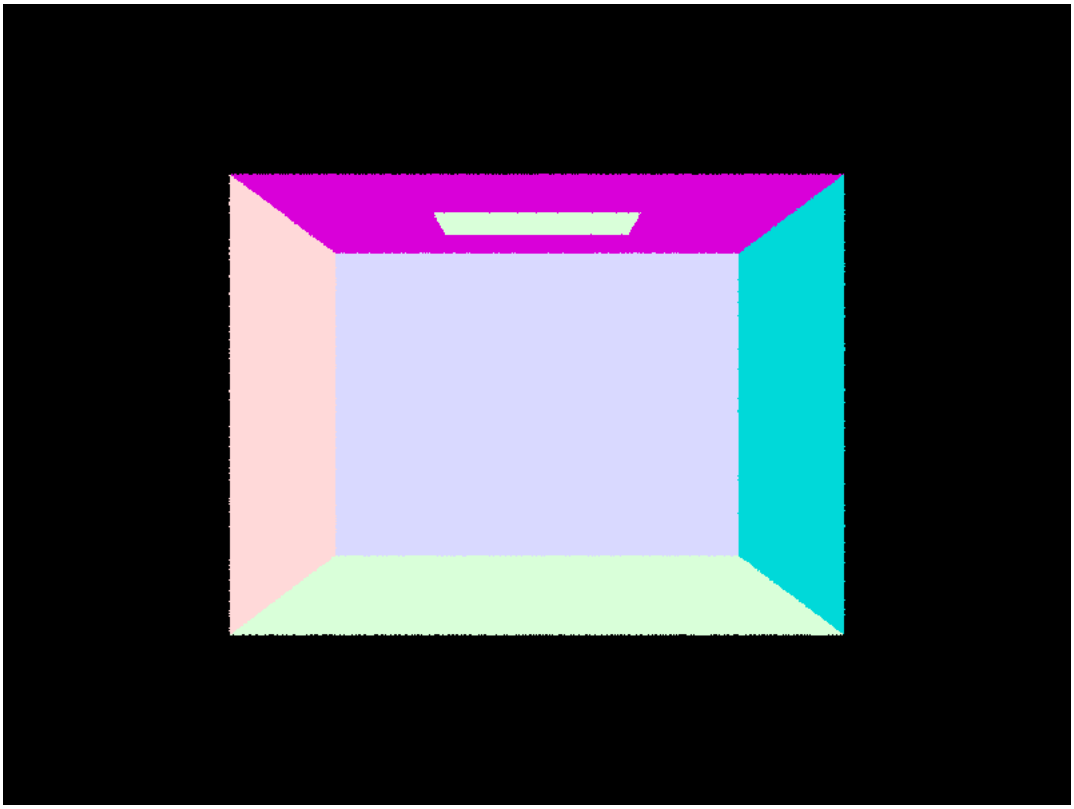
Normal shading with sampling at pixel center:



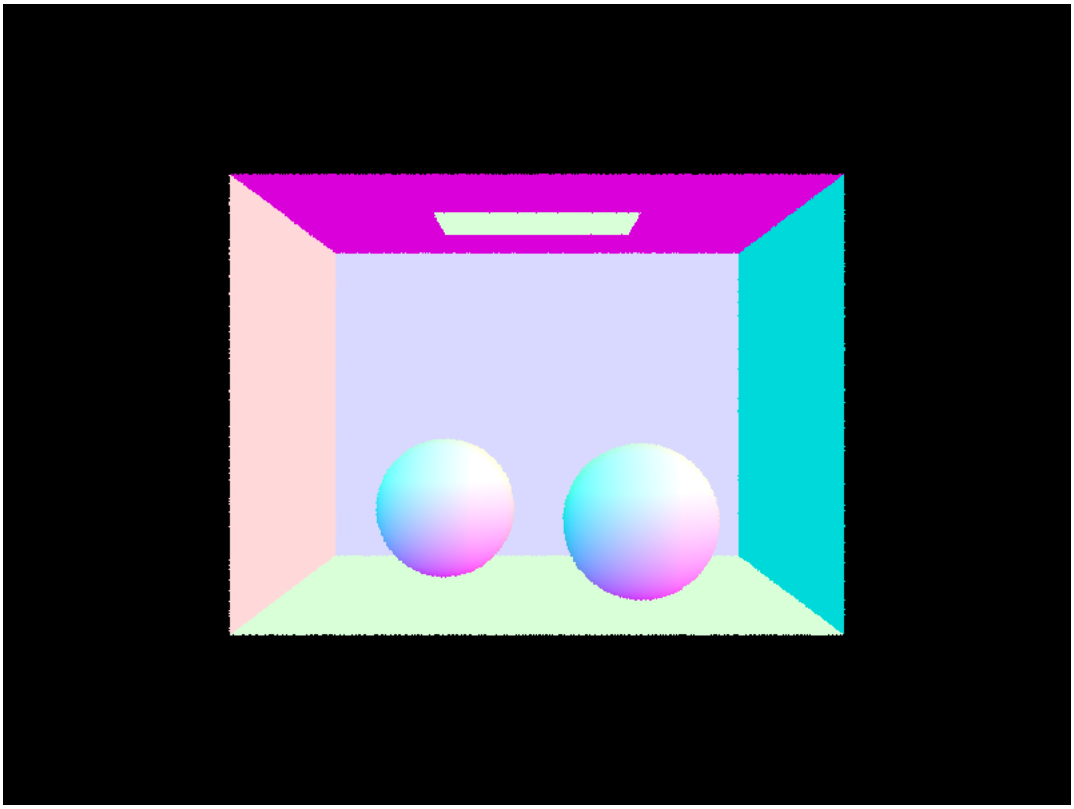
Normal shading with grid sampler



2.2 Task 3



2.3 Task 4



3 Part 2: Bounding Volume Hierarchy

The BVH construction algorithm is a recursive algorithm that partition the primitives into leaves of a binary tree. We use this data structure to accelerate the computation of ray intersection with object. In the beginning, the algorithm is performed on the root node. In each iteration, if the number of primitives in the node exceeds the maximum number allowed, the node is split into two halves. There are many heuristics to split the primitives. Here, we choose to split along the longest axis of the axis-aligned bounding box of the node. We separate the primitives according to the mean of a certain coordinate of the centroids of their bounding boxes. In the end, every internal node has a left and right child, while every leaf node has a list of primitives.

3.1 Task 1

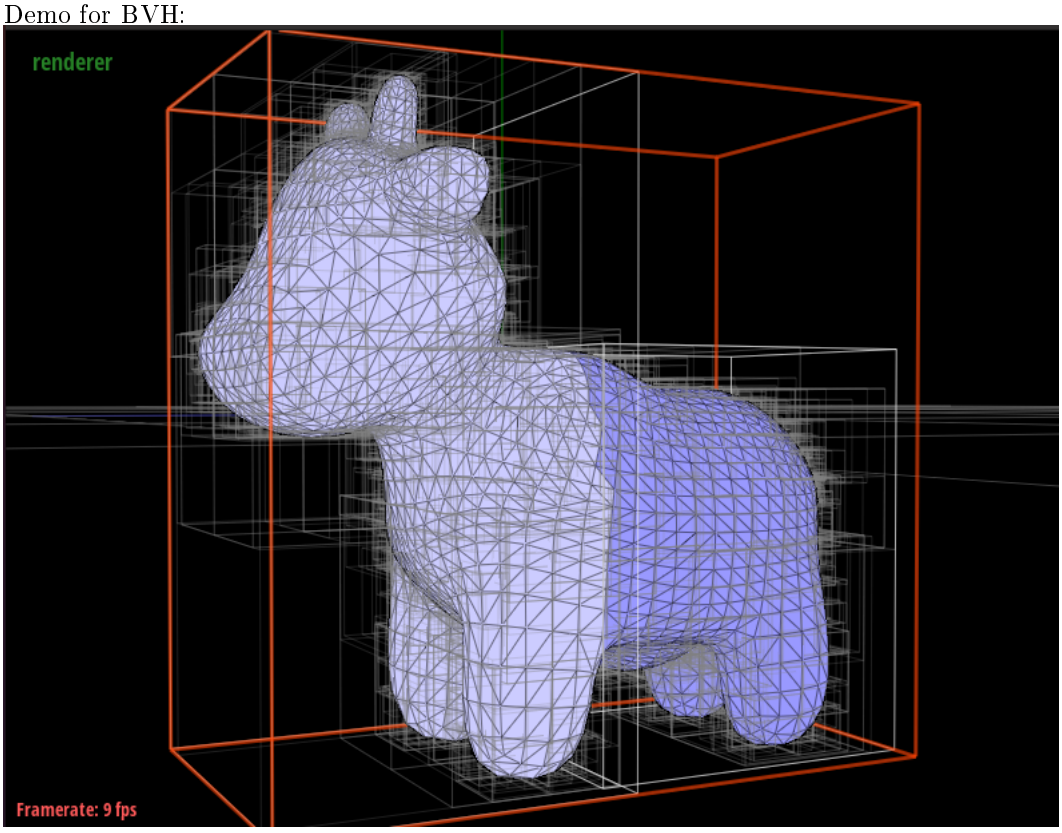
If the split point is chosen such that all primitives lie on only one side of the split point, then in each subsequent split, all the primitives will still lie on the same side, which leads to infinite recursive calls. As long as there is at least one primitive separated from the others, the algorithm will terminate in finite steps.

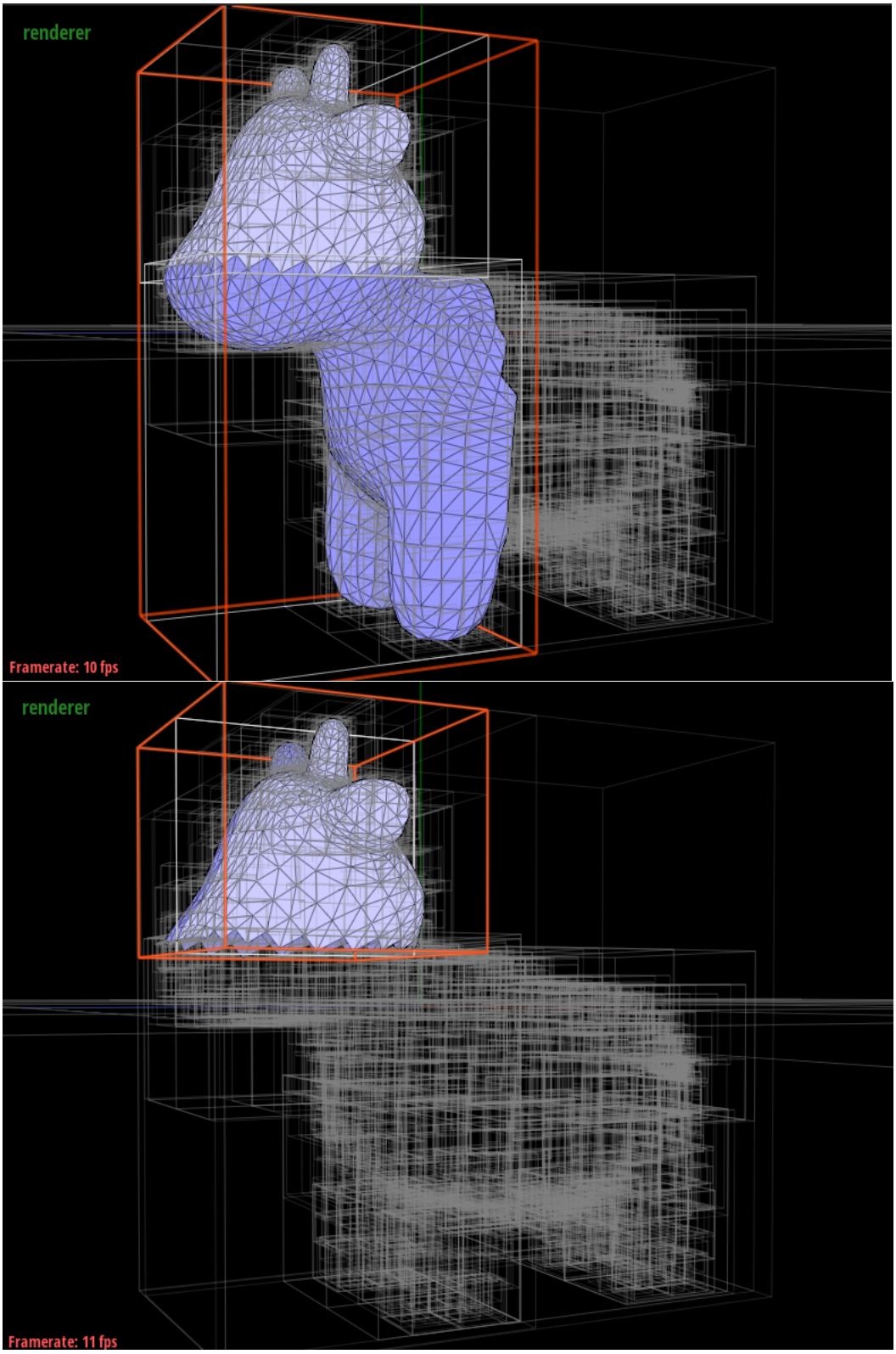
Acceleration for rendering cow.dae: 0.0002s + 47.2822s -> 0.0084s + 0.1611s

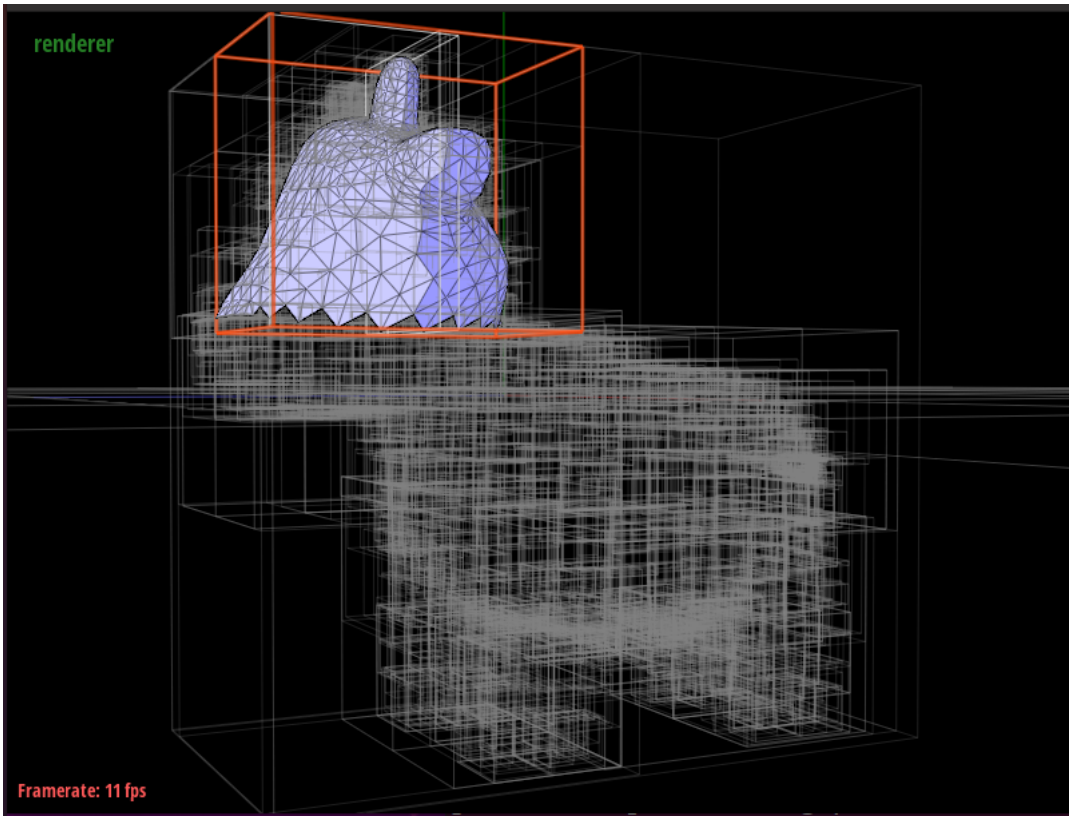
```
 ./pathtracer -t 8 -r 800 600 -f cow.png ../dae/meshedit/cow.dae
[PathTracer] Input scene file: ../dae/meshedit/cow.dae
[PathTracer] Rendering using 8 threads
[PathTracer] Collecting primitives... Done! (0.0006 sec)
[PathTracer] Building BVH from 5856 primitives... Done! (0.0002 sec)
[PathTracer] Rendering... 100%! (47.2822s)
[PathTracer] BVH traced 480000 rays.
[PathTracer] Average speed 0.0102 million rays per second.
[PathTracer] Averaged 5856.000000 intersection tests per ray.
[PathTracer] Saving to file: cow.png... Done!
[PathTracer] Job completed.
```

```
 ./pathtracer -t 8 -r 800 600 -f cow.png ../dae/meshedit/cow.dae
[PathTracer] Input scene file: ../dae/meshedit/cow.dae
[PathTracer] Rendering using 8 threads
[PathTracer] Collecting primitives... Done! (0.0006 sec)
[PathTracer] Building BVH from 5856 primitives... Done! (0.0084 sec)
[PathTracer] Rendering... 100%! (0.1611s)
[PathTracer] BVH traced 480000 rays.
[PathTracer] Average speed 2.9800 million rays per second.
[PathTracer] Averaged 3.434498 intersection tests per ray.
[PathTracer] Saving to file: cow.png... Done!
[PathTracer] Job completed.
```

Demo for BVH:




renderer
Framerate: 10 fps
renderer
Framerate: 11 fps



3.2 Task 3

For BVHAccel::intersect(), the bound box of the nearest intersection must be intersected by the ray. Therefore, if we check every primitive in every BBox intersected, we must find the nearest intersection.

In the BVH intersection functions, the intersection functions of corresponding primitives are called. Therefore, if all primitives update i and r.max_t correctly in their own intersection functions, the BVH intersection function does not need to update them.

Scene	w/o BVH Time (s)	w BVH Time (s)
cow	47.2824	0.1695
maxplanck	539.9757	0.2786
Cblucy	1887.0623	0.2509

As shown in the above chart, the rendering speed of cow, maxplanck, CBlucy are increased 280, 2000, and 7500 times.

4 Part 3: Direct Illumination

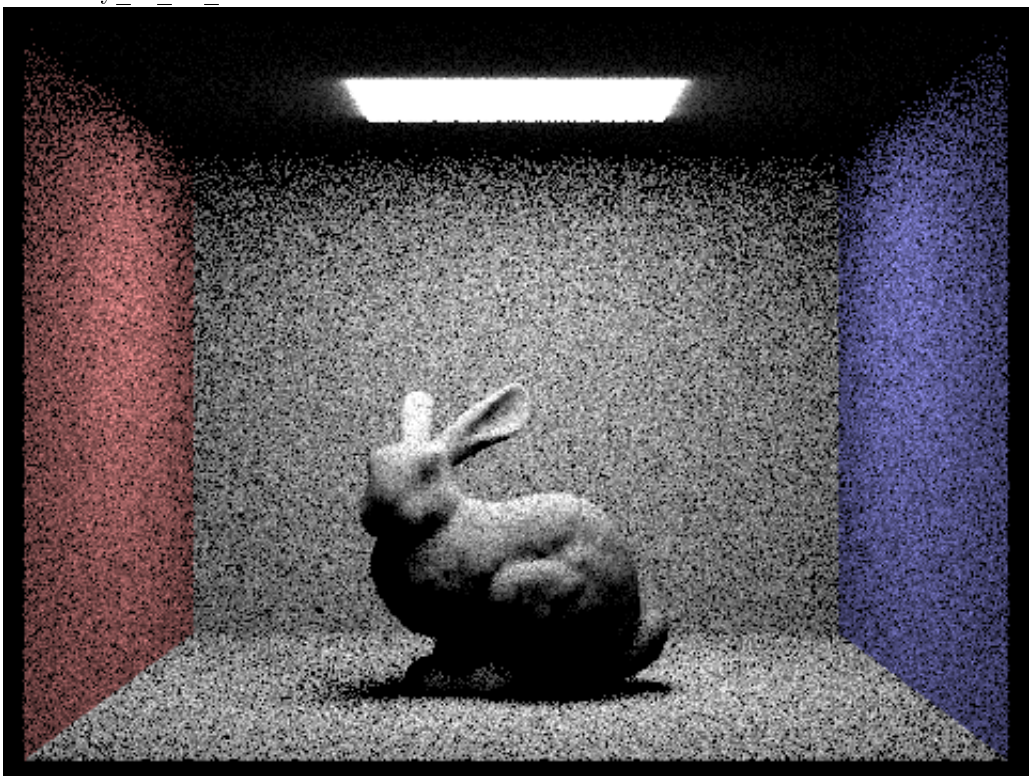
estimate_direct_lighting_hemisphere() calculates the intersection of a ray (from the camera) with the scene first. Then, at the intersection point, it samples over the hemisphere using a hemisphere sampler. It then calculates the intersection of the sampling ray with the scene. If there is an intersection, it gets the emission of the intersection. The emission is weighted and added to

the total radiance towards the pixel. In the end, we use the total radiance information to get the color of the pixel.

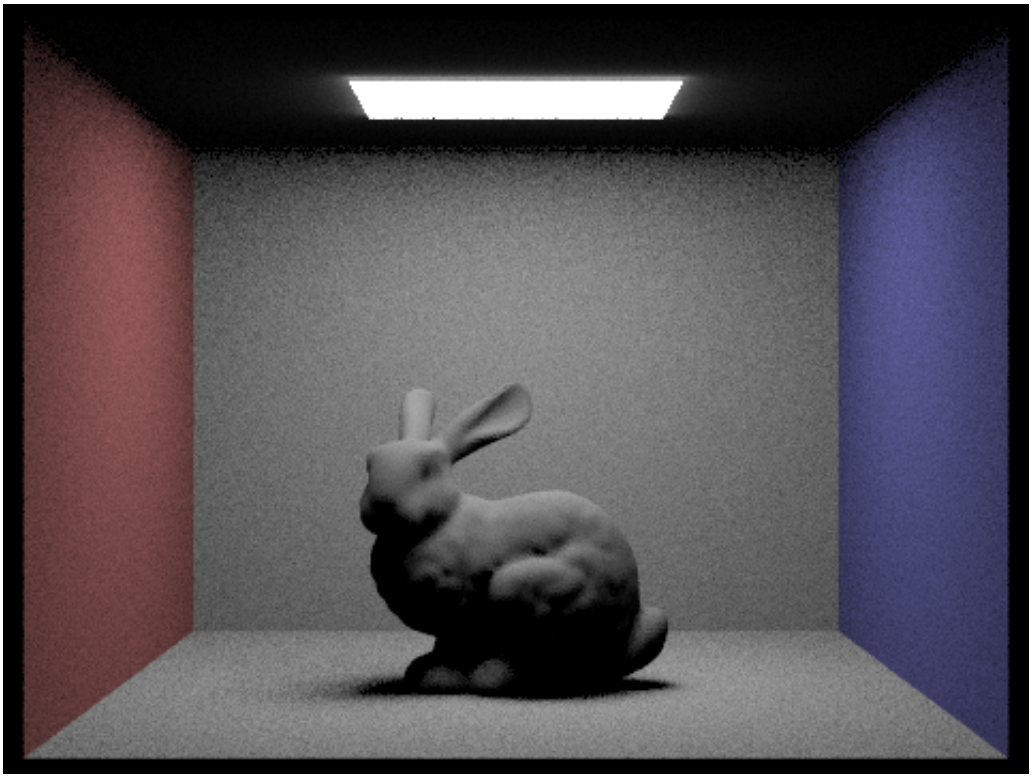
estimate_direct_lighting_importance() samples differently at the intersection point. Instead of casting rays uniformly over the hemisphere, it samples directly the light sources. In this way, the average radiance of the sampling rays are much higher. All the light sources are guaranteed to be sampled so that there are less noise.

4.1 Task 3

CBbunny_H_16_8:

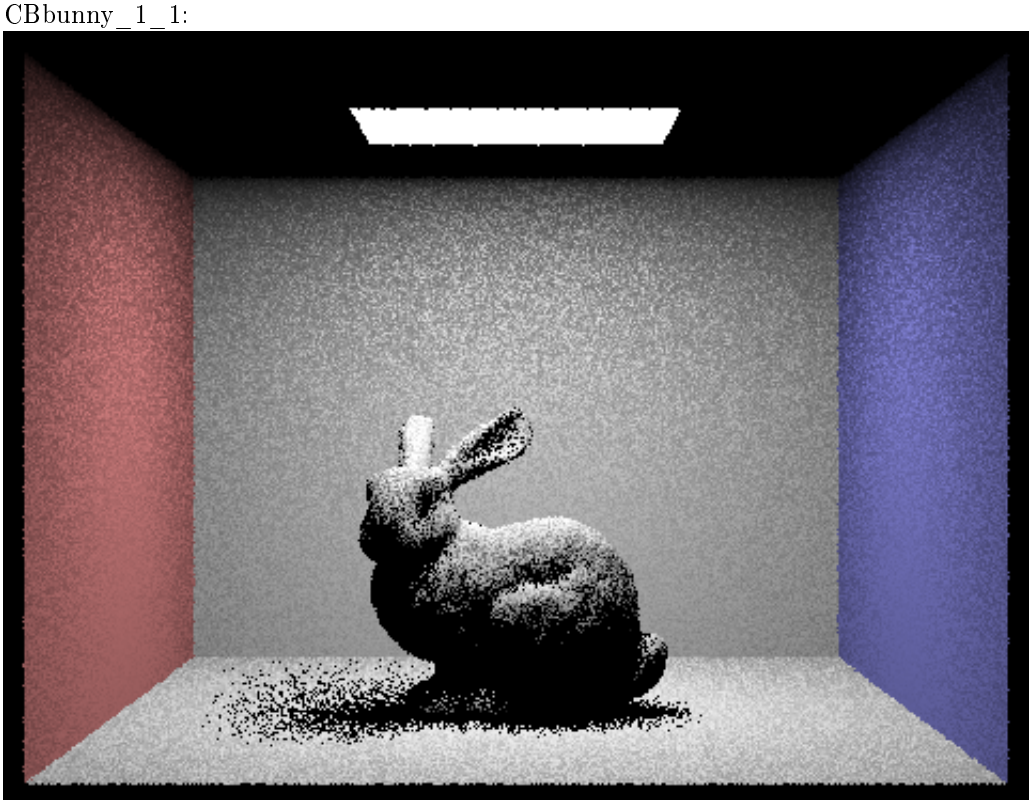


CBbunny_H_64_32:

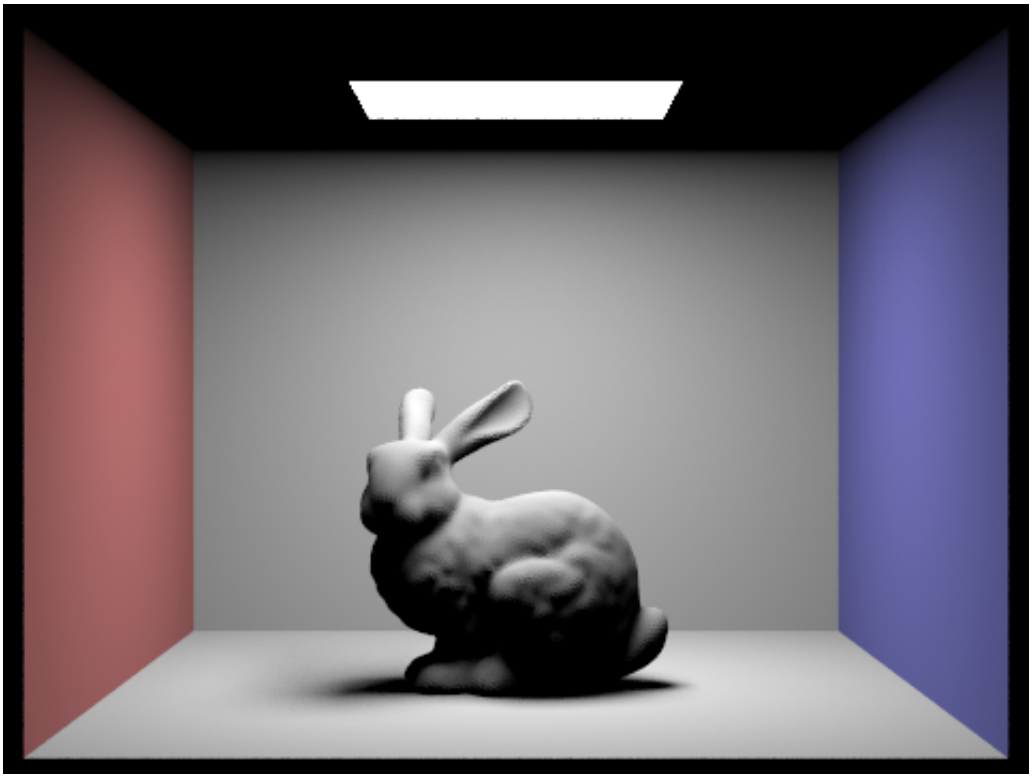


4.2 Task 4

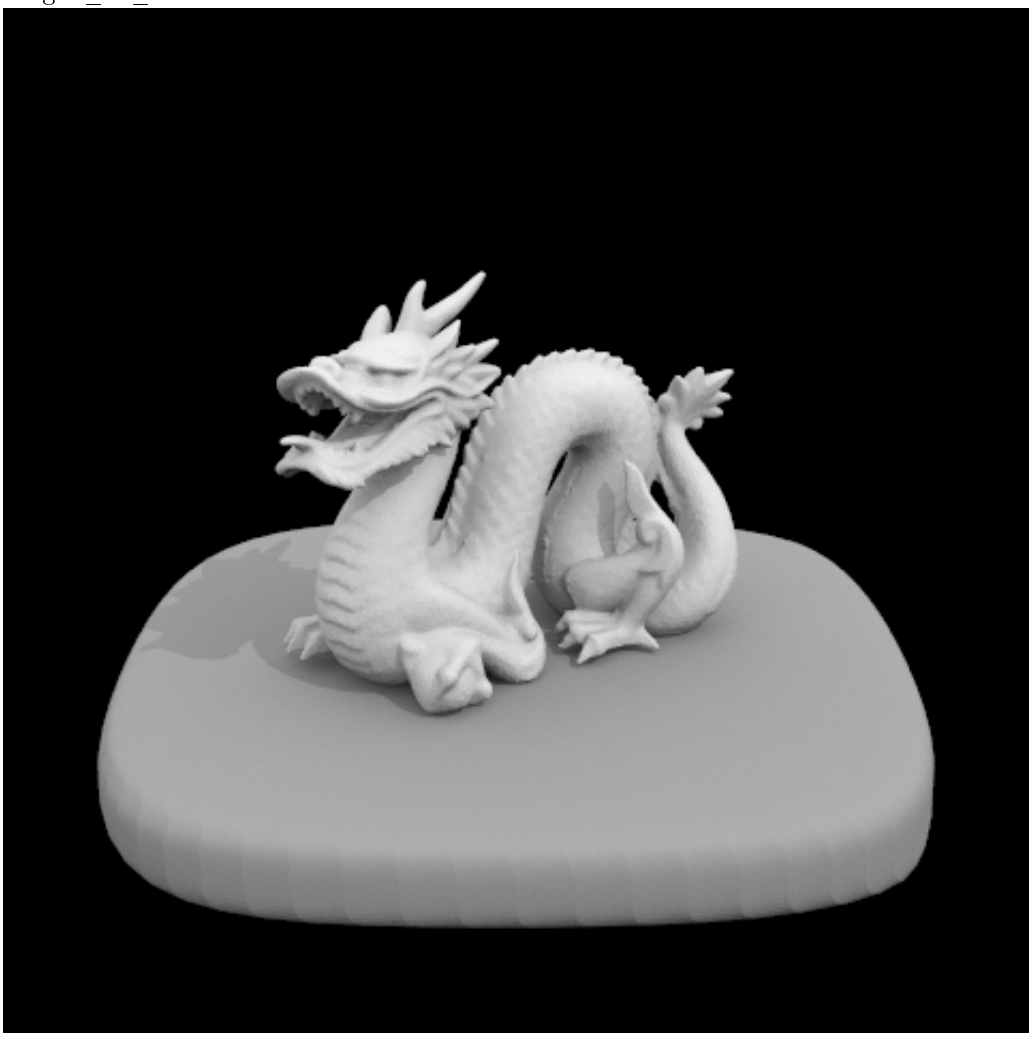
CBbunny_1_1:



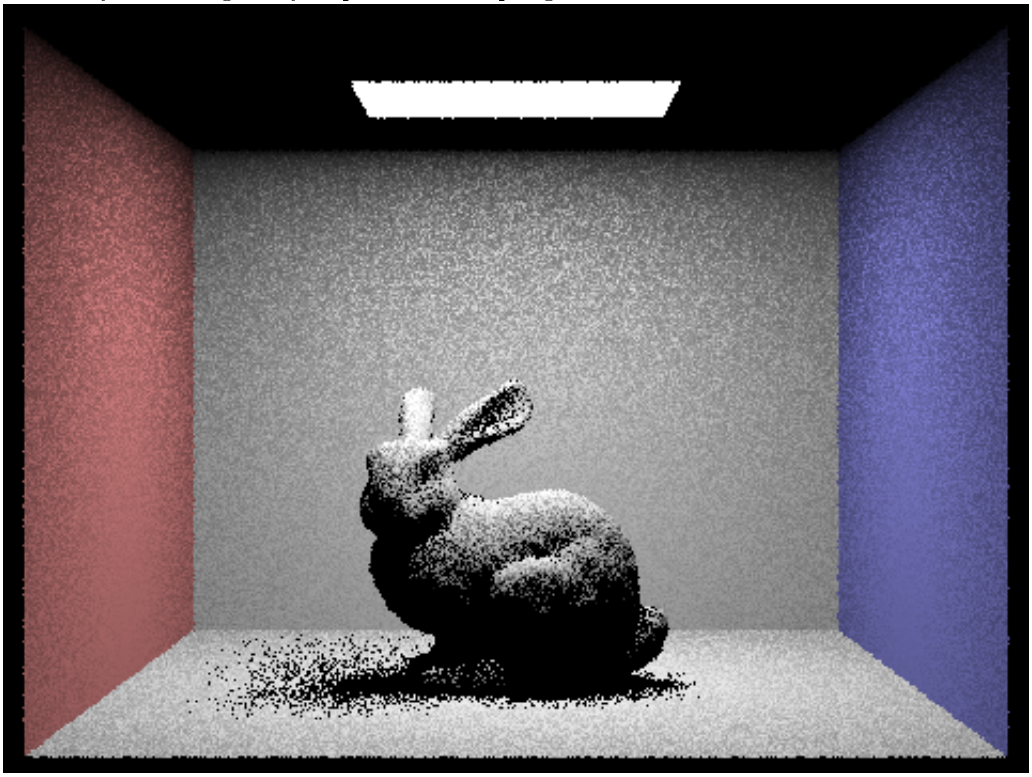
CBbunny_64_32:



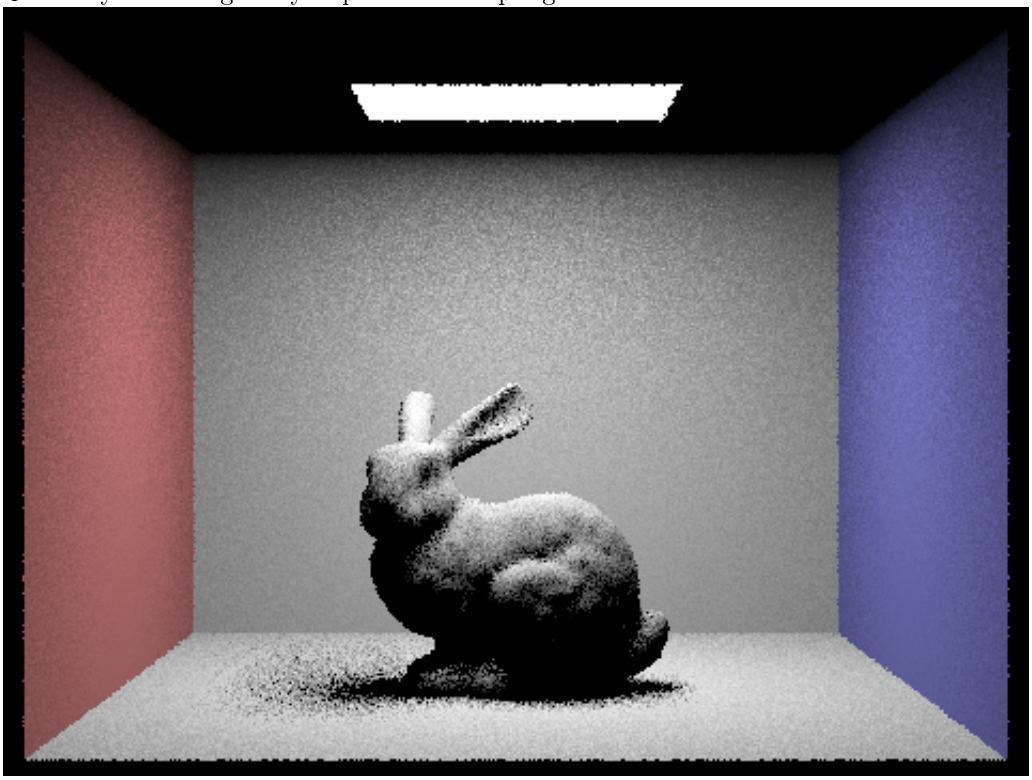
dragon_64_32:



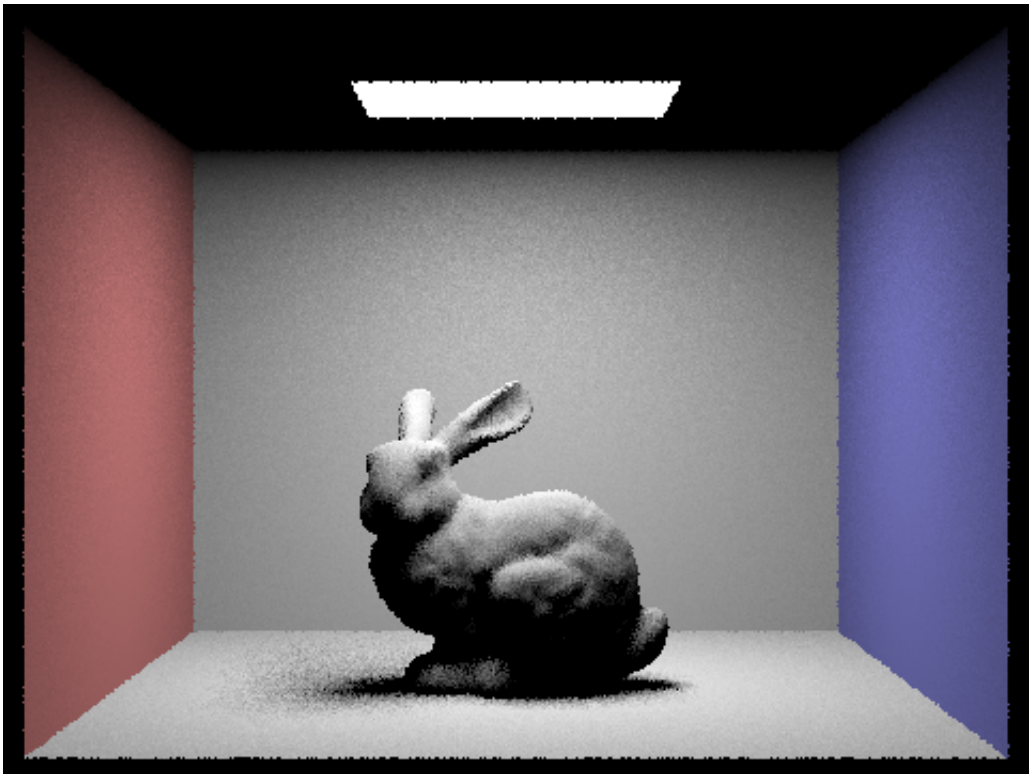
CBbunny with 1 light ray importance sampling:



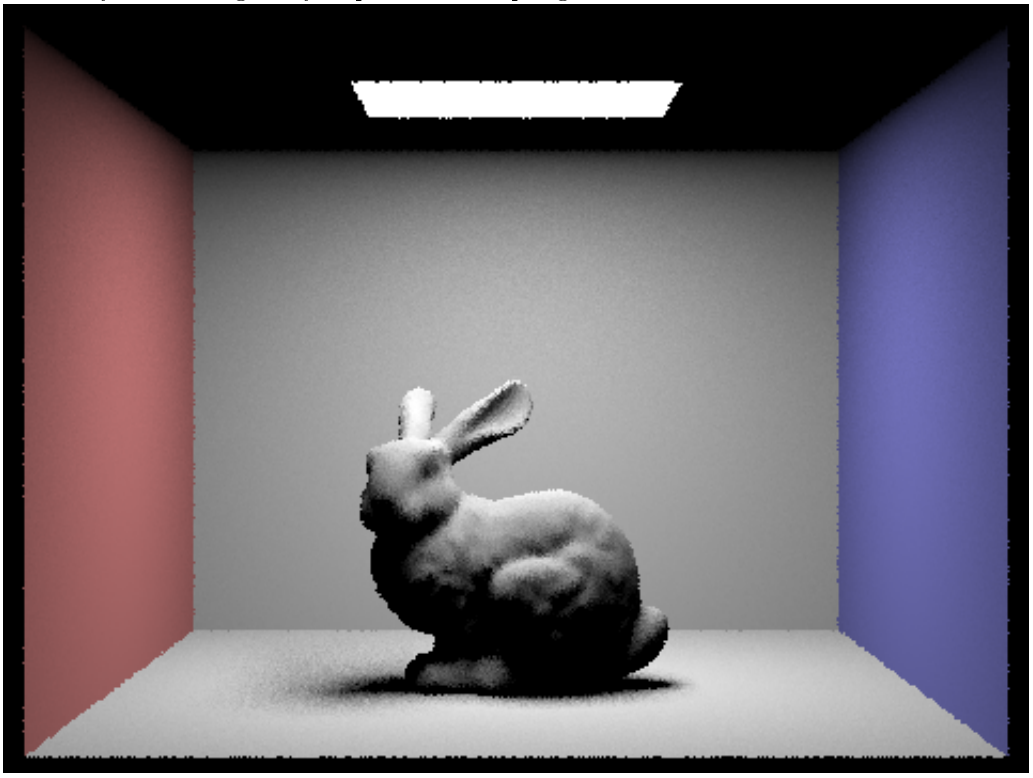
CBbunny with 4 light ray importance sampling:



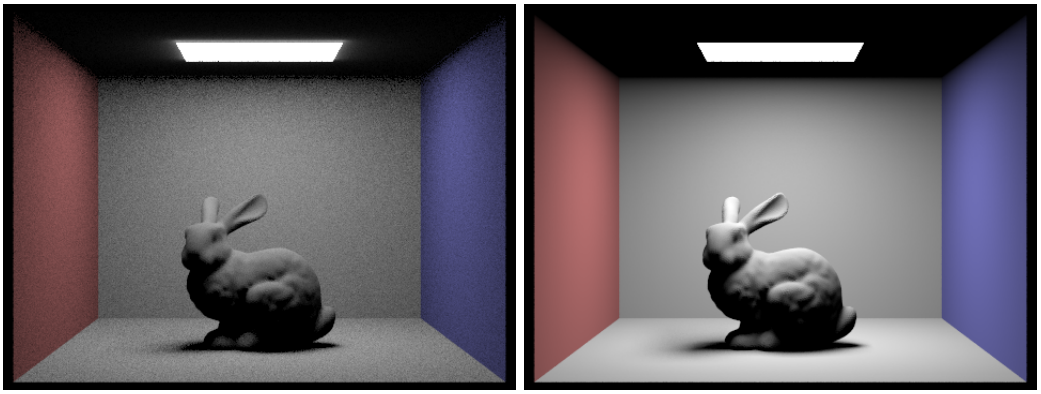
CBbunny with 16 light ray importance sampling:



CBbunny with 64 light ray importance sampling:



Comparison between hemisphere and importance sampling:

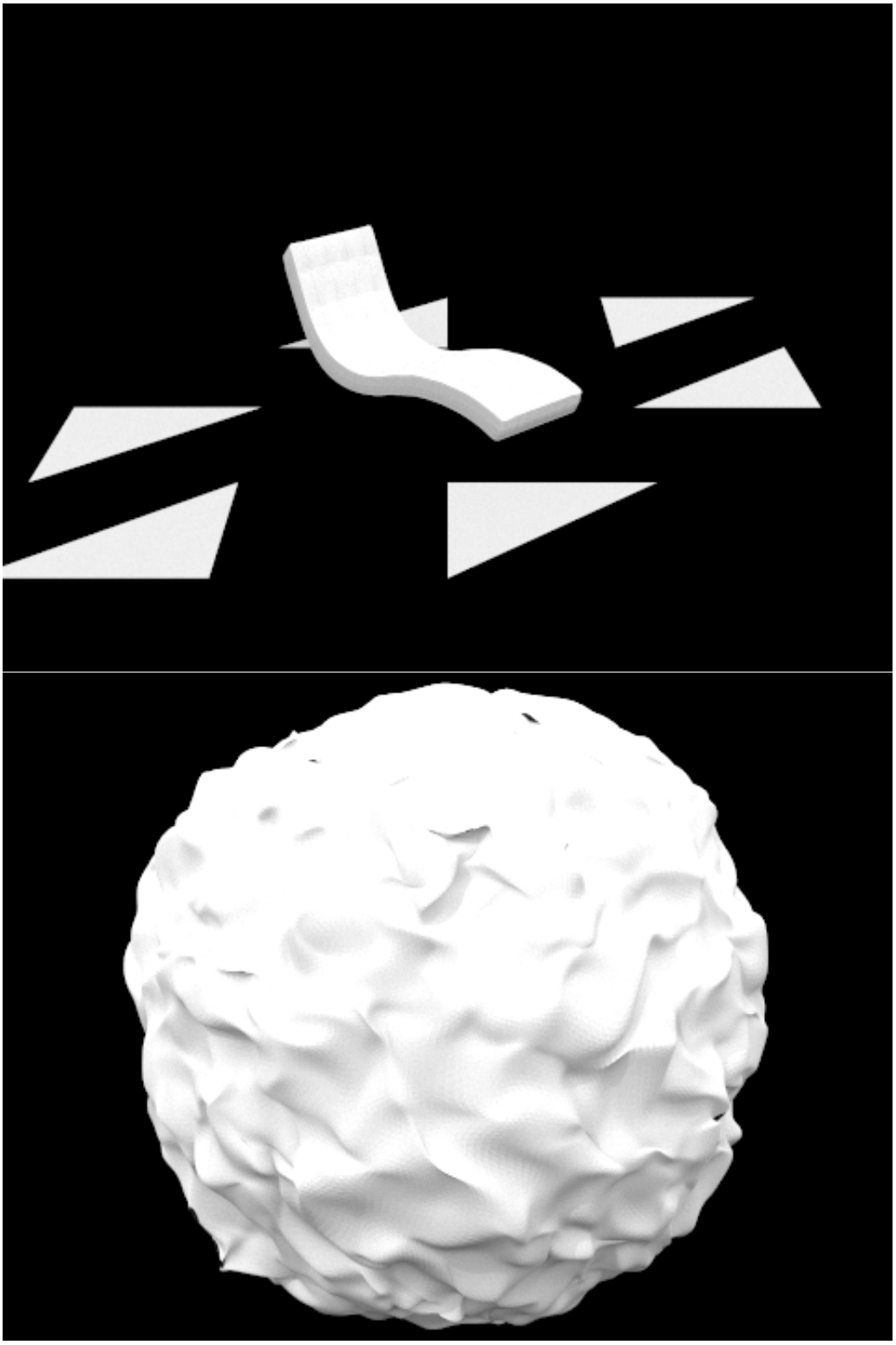


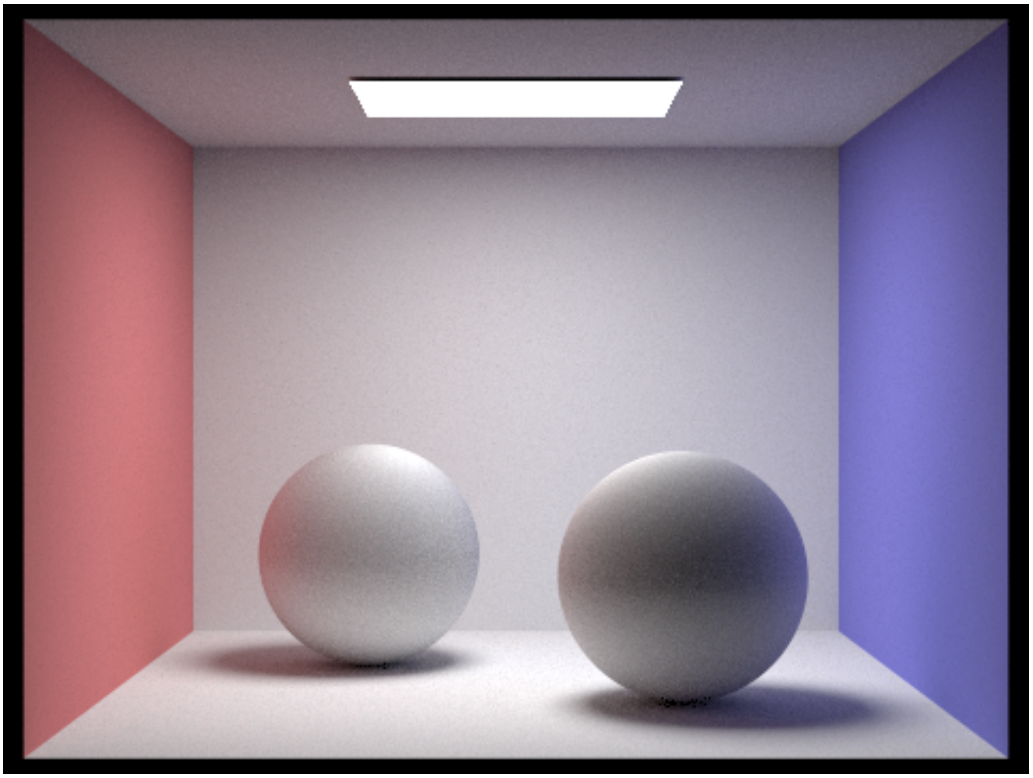
As shown in the above figures, with the same parameters, importance sampling significantly reduces noises in the result. In hemisphere sampling, the sampling ray has a very small probability to hit the light source. Therefore, neighboring pixels can have very different number of sampling rays that hit the light source. Some pixels that should have been illuminated may not even sample the light source. Hence, the result is very noisy. Importance sampling makes sure that every pixel samples the light source evenly.

5 Part 4: Global Illumination

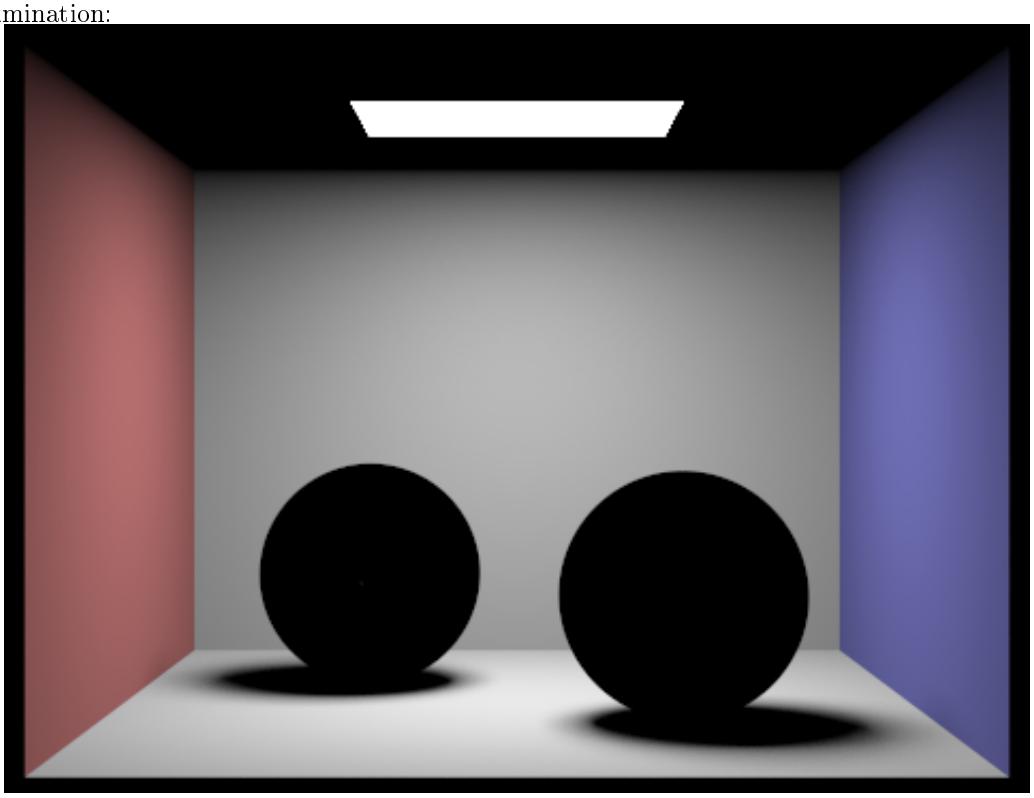
We implement a recursive function at_least_one_bounce_radiance() to estimate global illumination (except emission, which is estimated in zero_bounce_radiance()). This function firstly calculates the hit point and calls one_bounce_radiance() to get the direct illumination (except emission) on it. Then, if the maximum ray depth is not reached and the Russian Roulette is passed (if this is the first-level ray, the Russian Roulette is ignored), it randomly casts a ray from the hit point. If the new ray hits the scene, it calls itself to evaluate the radiance coming from that new hit point. This indirect radiance is weighted and added to the radiance of the camera ray.

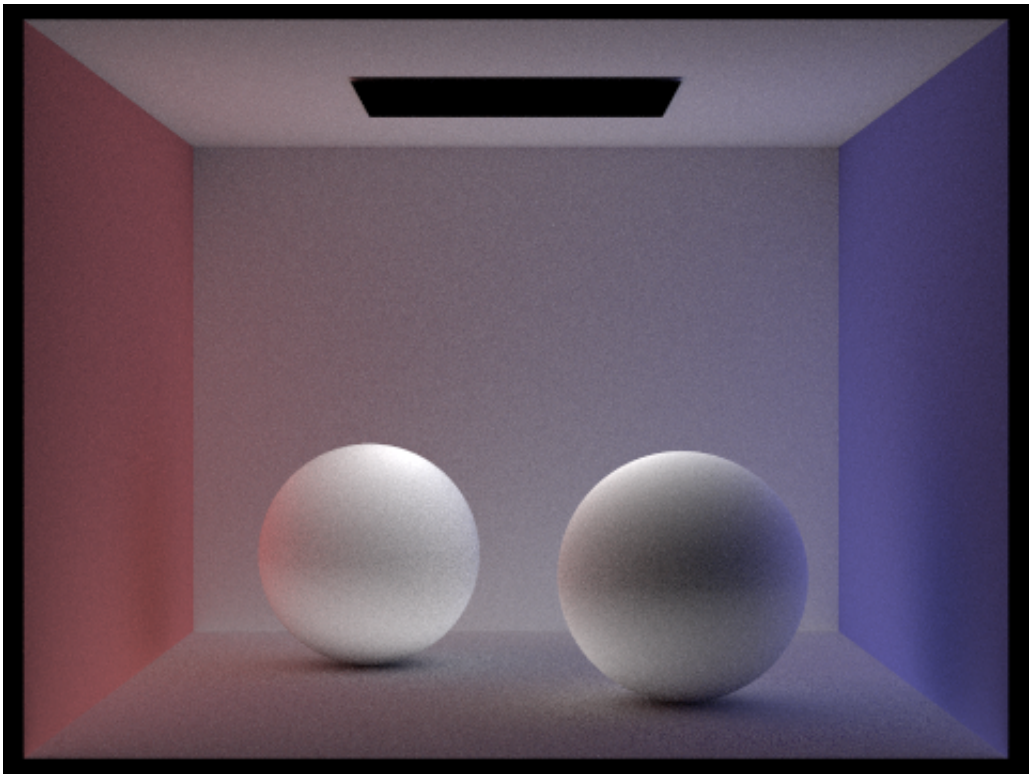
Show some images rendered with global (direct and indirect) illumination. Use 1024 samples per pixel:





Pick one scene and compare rendered views first with only direct illumination, then only indirect illumination:

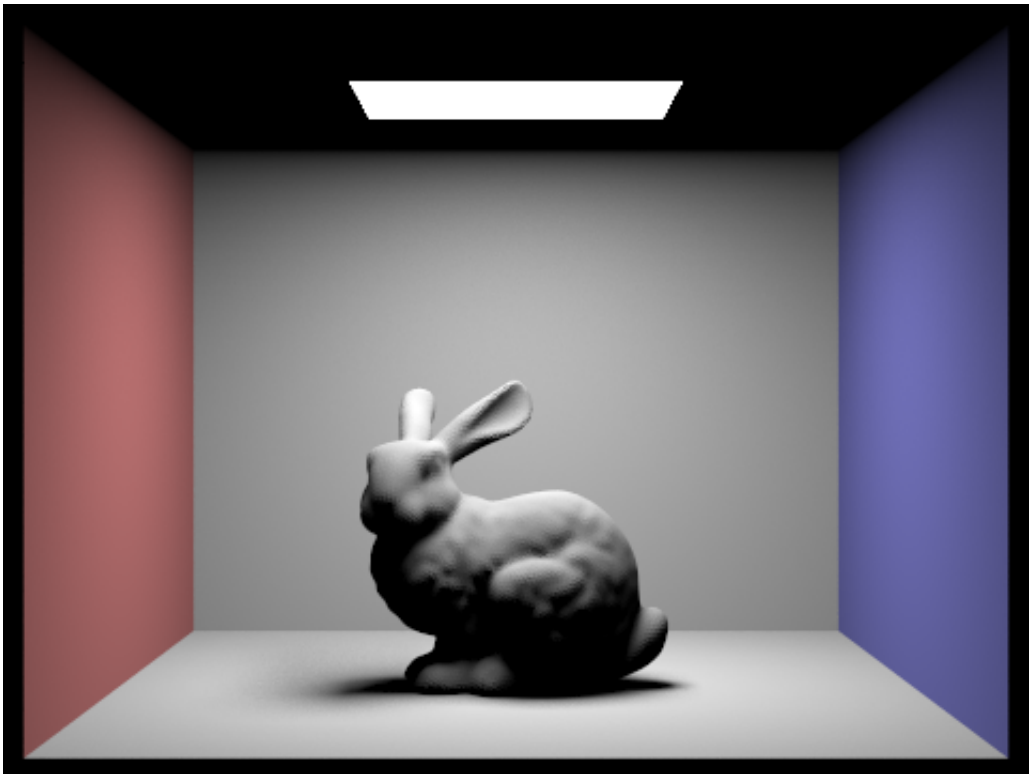




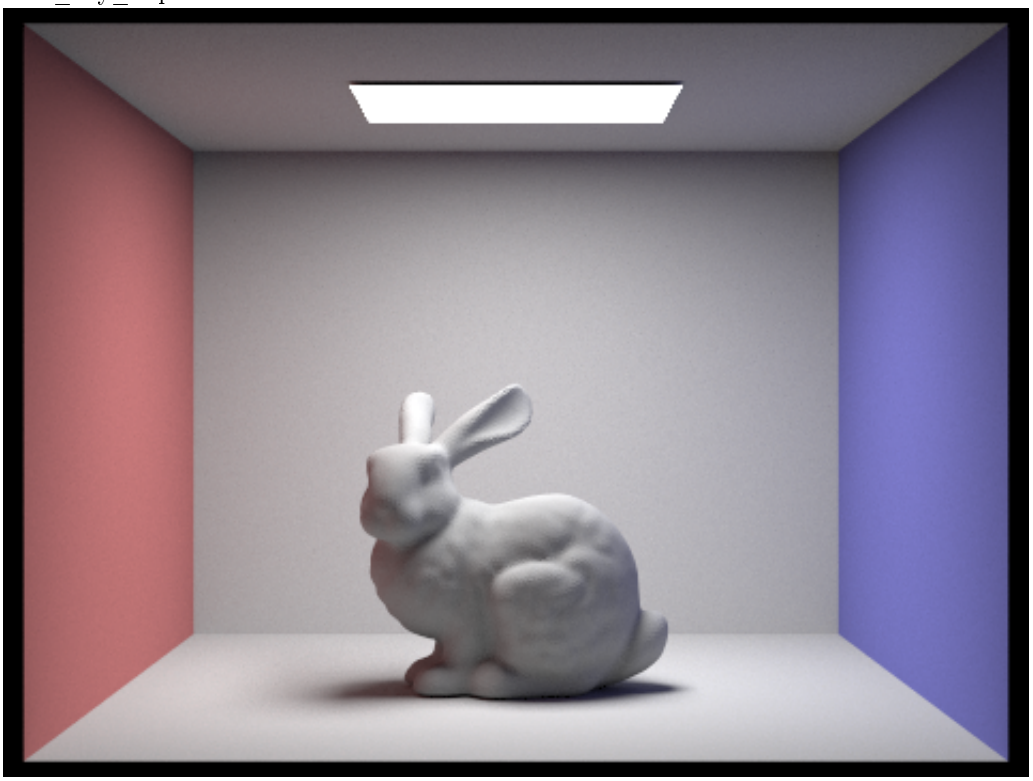
For CBbunny.dae, compare rendered views with max_ray_depth set to 0, 1, 2, 3, and 100:

max_ray_depth = 0:

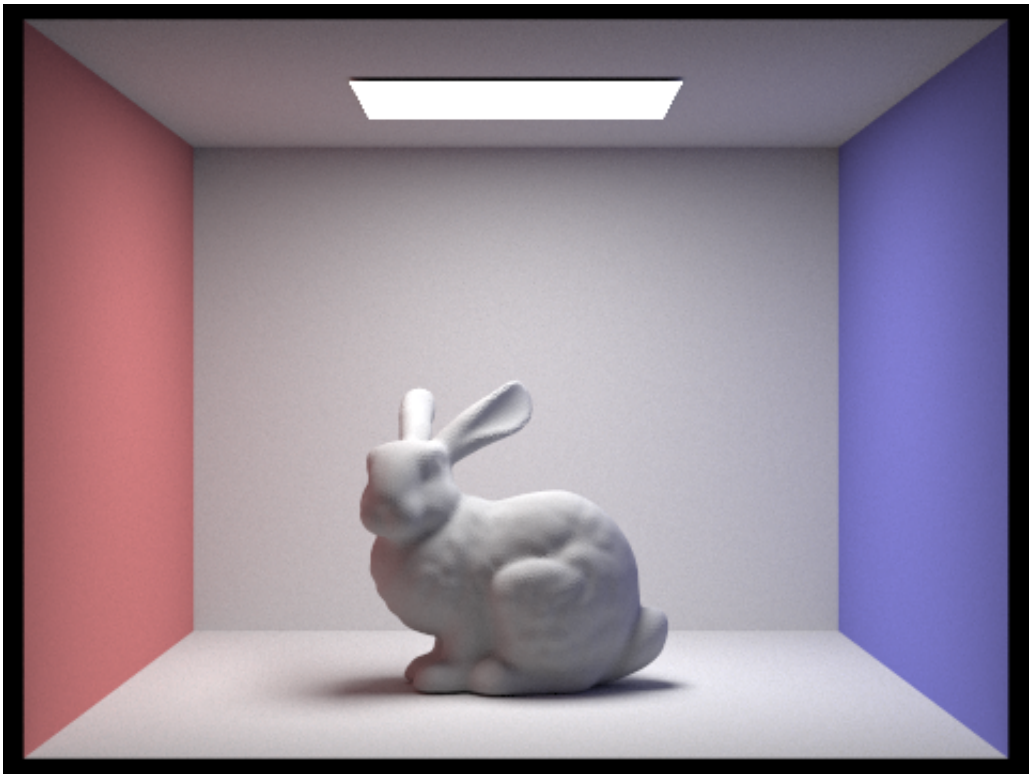
max_ray_depth = 1:



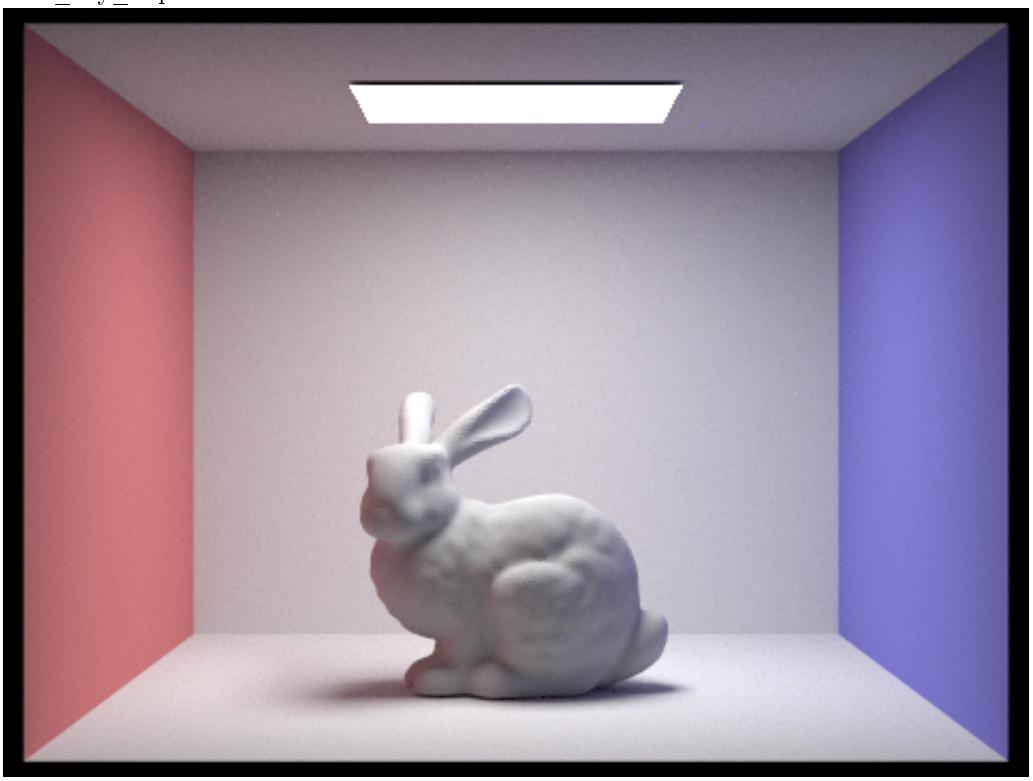
max_ray_depth = 2:



max_ray_depth = 3:

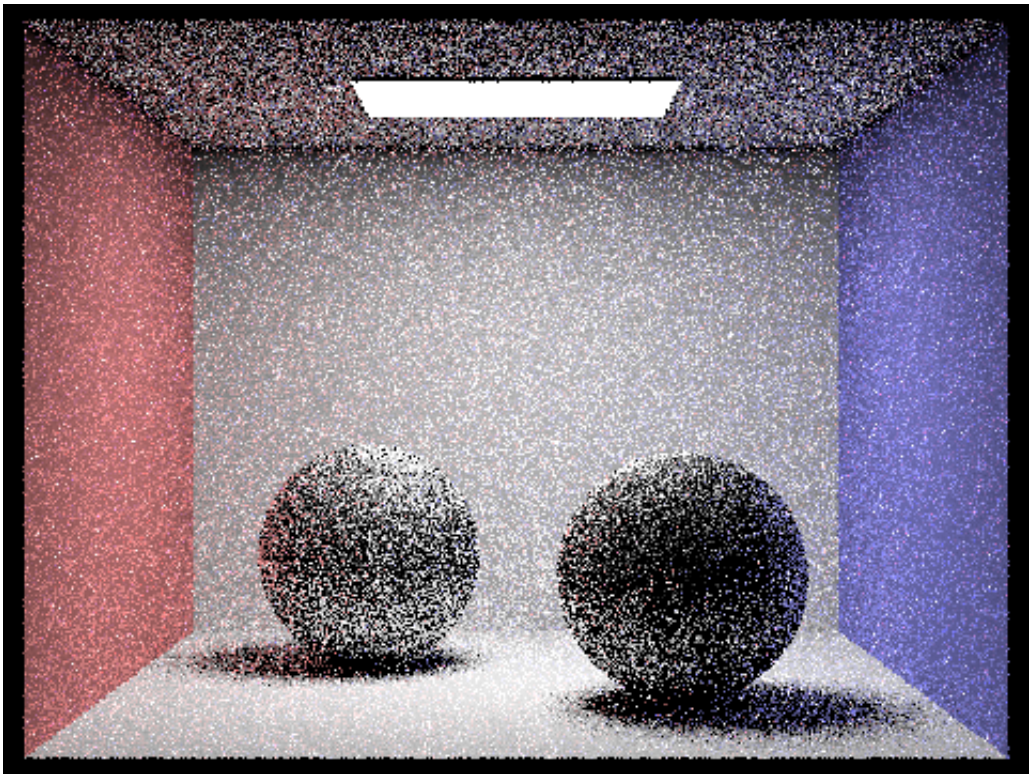


max_ray_depth = 100:

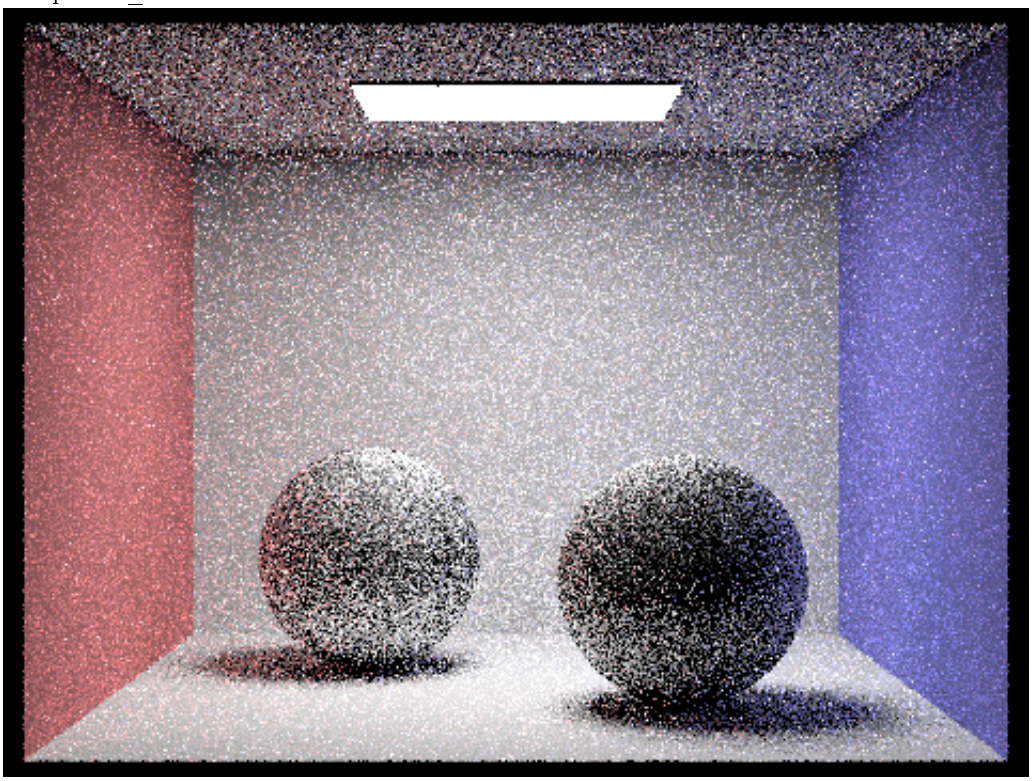


Pick one scene and compare rendered views with various sample-per-pixel rates, including at least 1, 2, 4, 8, 16, 64, and 1024. Use 4 light rays:

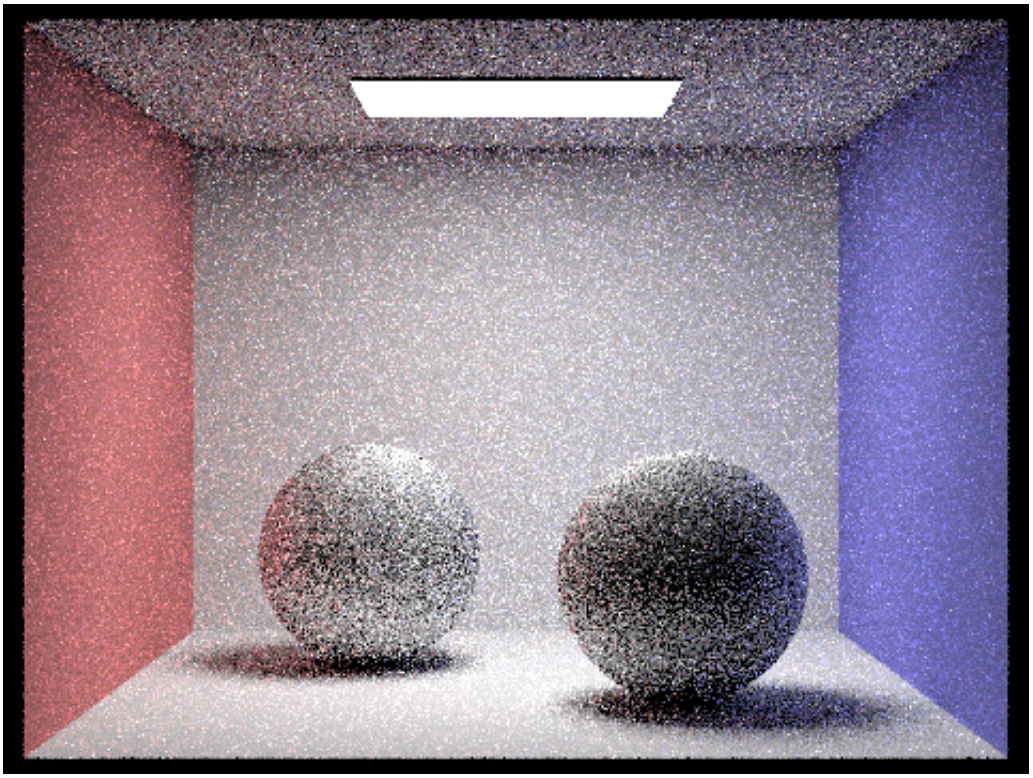
CBspheres_lambertian s=1:



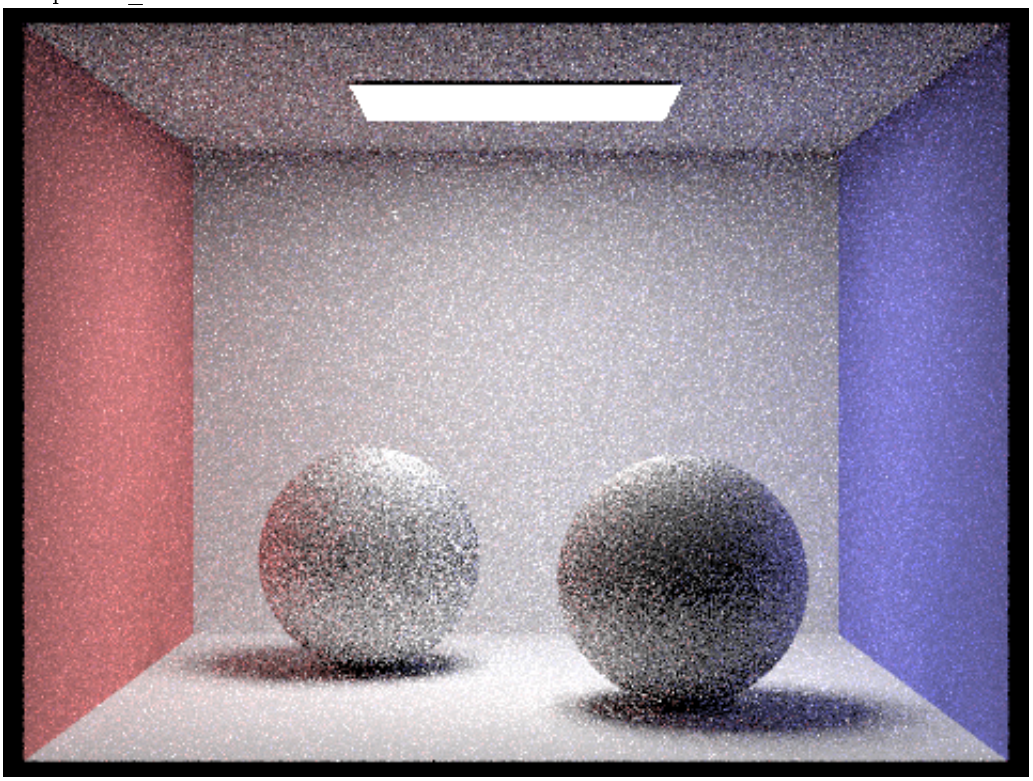
CBspheres_lambertian s=2:



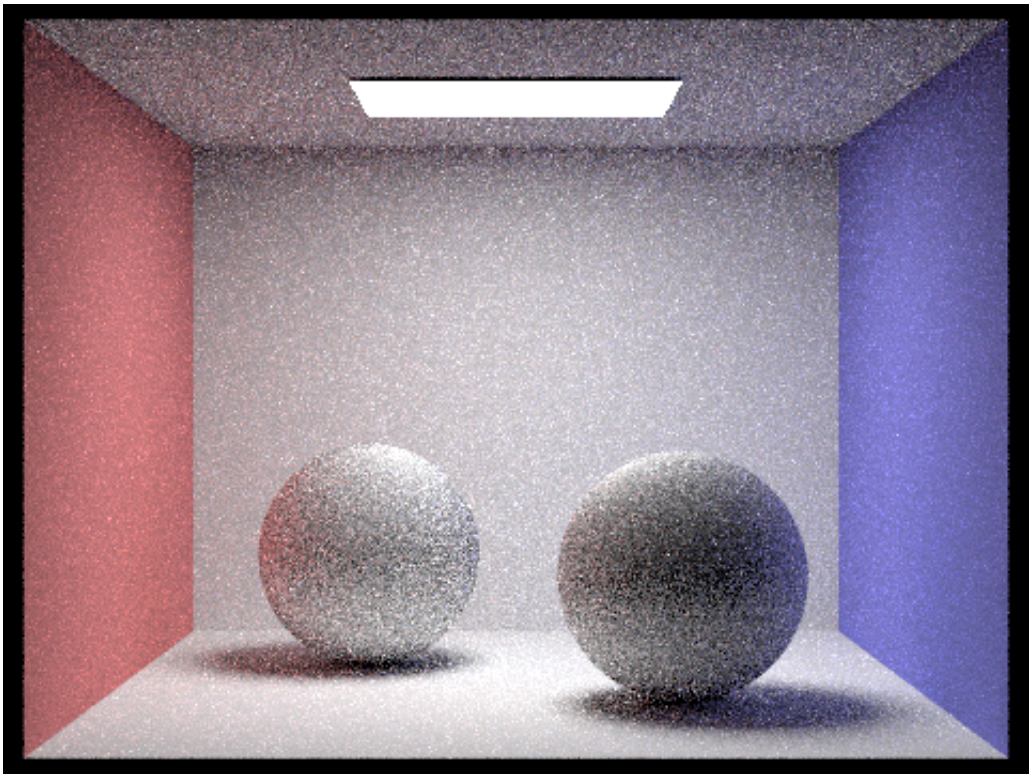
CBspheres_lambertian s=4:



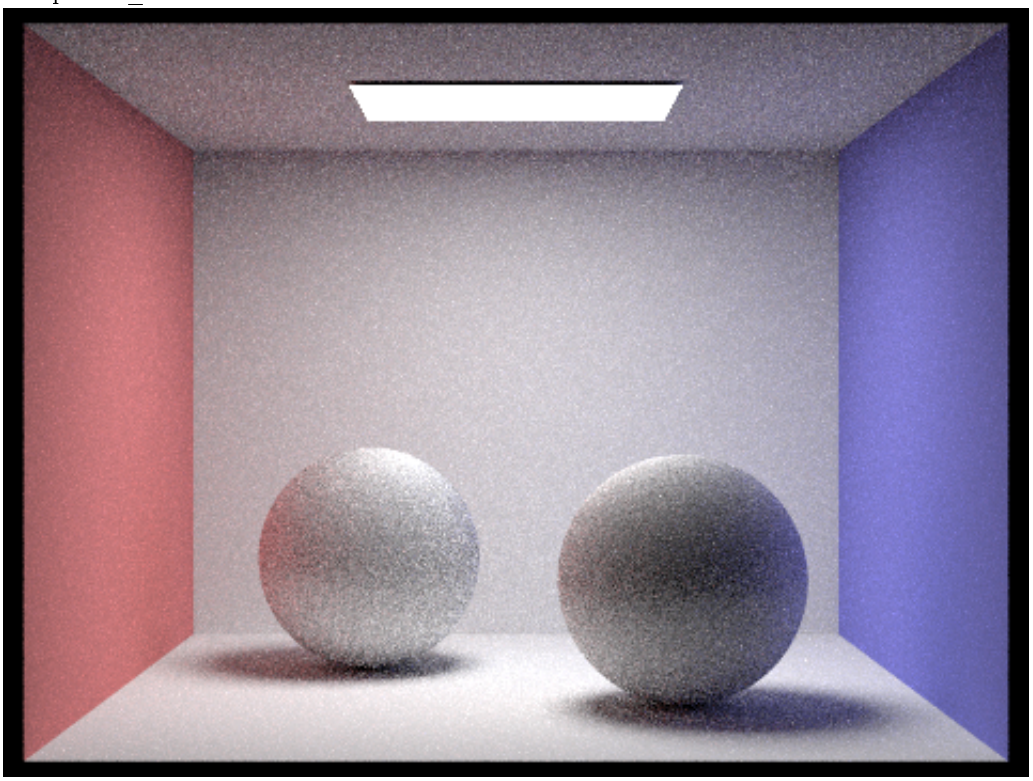
CBspheres_lambertian s=8:



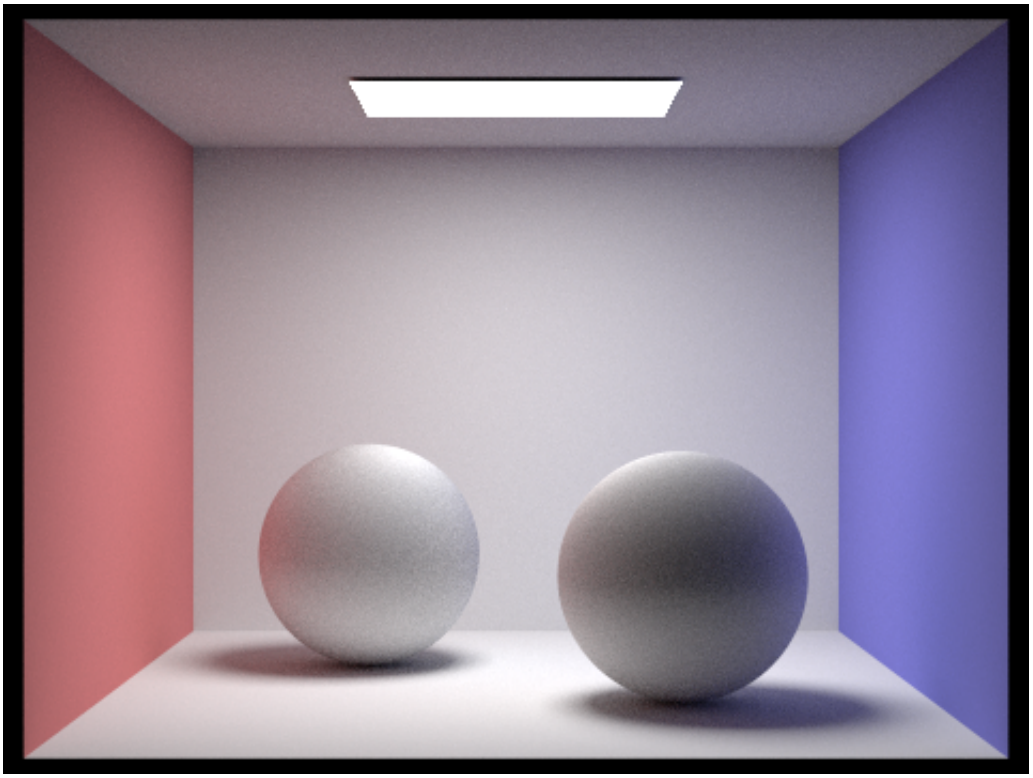
CBspheres_lambertian s=16:



CBspheres_lambertian s=64:



CBspheres_lambertian s=1024:

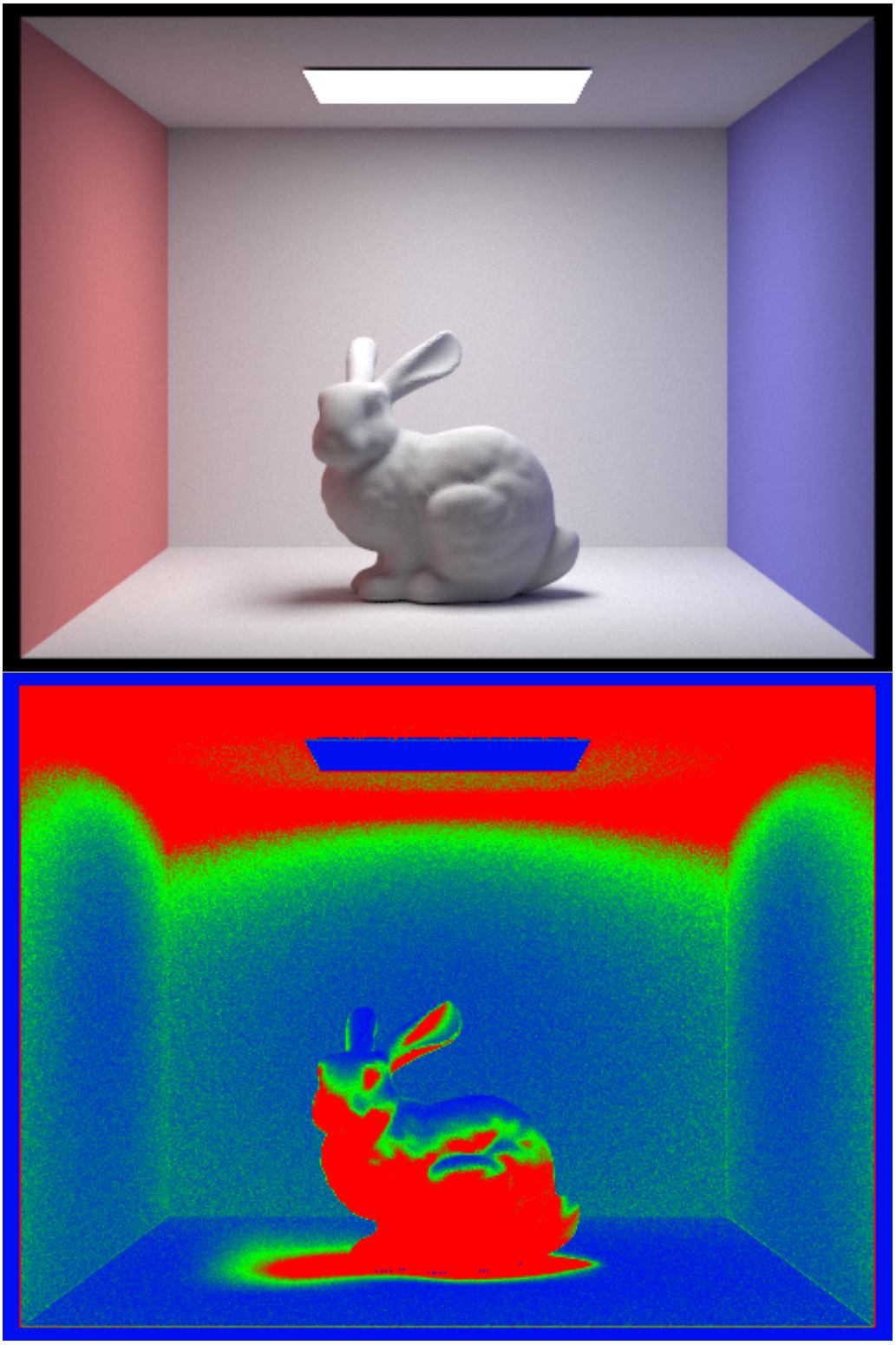


6 Part 5: Adaptive Sampling

In the scene, some pixels converge faster while others need more iterations. Adaptive sampling stops further sampling for nearly converged pixels, effectively reducing rendering time.

When sampling each pixel, we keep a running sum of illumination as well as its square. We also keep a counter to count the actual number of samples. After every batch of samples, we use the mean and variance of the samples to check for convergence. If the convergence satisfy the tolerance, we stop further sampling. This method does not affect the quality of the result much obviously but reduces rendering time effectively.

CBbunny with 2048 samples per pixel:



7 Extra

Implement a more memory efficient BVH by storing all the Primitive pointers in one large vector and only keeping index references into small contiguous chunks of this vector inside the leaves. Alternatively, think of and research other ways to compress the tree. Compare memory

usage to the original data structure.

In my construct_bvh() implementation, there is no new() constructs. In other words, construct_bvh() does not use any extra memory to store the Primitive pointers. Instead, it sorts the original large vector of Primitive pointers. Theoretically, this methods saves O(NlogN) memory compared to creating a new vector for each node, where N is the number of primitives.