Assignment 3: PathTracer

Shenao Zhang

Part 1: Ray Generation and Intersection

In this part, what we mainly do is generate ray through every pixel and intersect the rays and the triangles, spheres. To implement it, we have the following four tasks.

Task 1:Filling in the sample loop

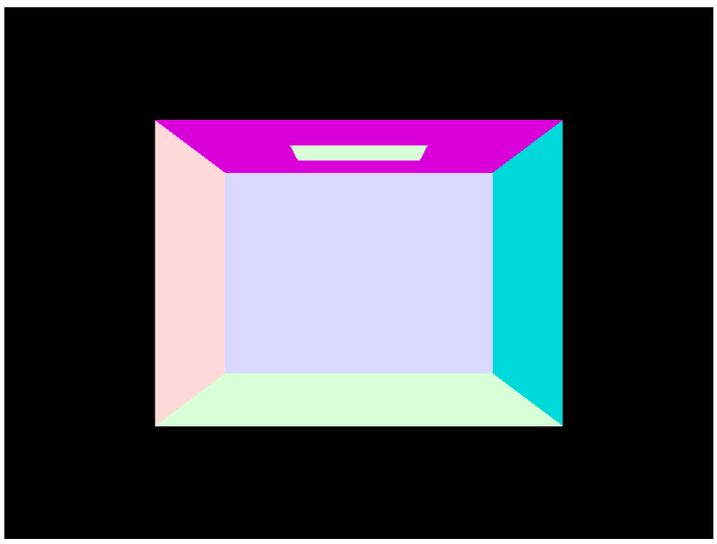
In this task, we calculate the integral of the irradiance over every pixel to futher after generating camera rays, determining which triangles every ray intersects with.
When the sample rate(samples per pixel) equals to 1, which means only one simulated ray from camera is generated from the center of one pixel. When the sample rate is greater than 1, there are multiple random rays from camera generated from this pixel. Accumulate spectrum of every ray for **num_samples** times. Then we calculate the average of the accumulated spectrum as the spectrum of the pixel. The rays can be generated by function **camera->generate_ray** and the random sampling points can be determined by function **random**.
Then we need to generate camera rays.

Task 2:Generating camera rays

For a camera, assume it is the origin, and it looke along the **-z** axis. So we can use two given field of view angle **hFov**, **vFov** to define bottem left and top right corner of a sensor plane. Then we can define the x and y coordinate in the sensor plane.(e.g. bottem_left.x+(top_right.x-bottem_left.x)*x). This is the direction in camera space. Convert this vector to world space using matrix transform **c2w**. Finally, **ray.o** equlas to the camera's position **pos**, **ray.d** equals to the normalized world space vector.
After generating rays through sample points from every pixel, we then determine which triangle the rays are intersecting.

Task 3:Intersecting Triangles

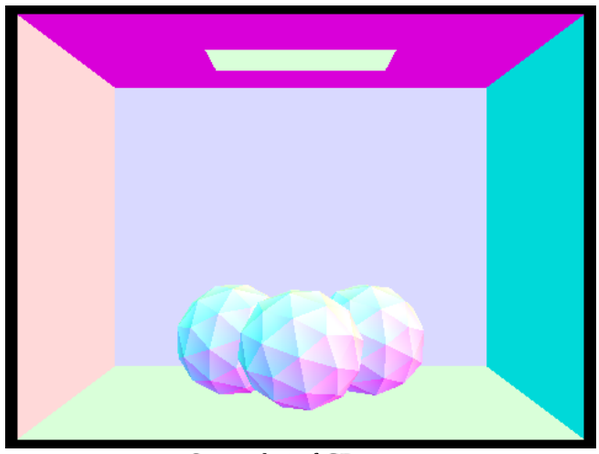
To find the intersecting triangle, there are many methods. Here, I use Möller-Trumbore algorithm. It is a fast ray-triangle intersection algorithm. This algorithm takes advantage of the intersection point in terms of barycentric coordinates. Let me briefly introduce how the algorithm works.
First write a point P inside a triangle(ABC) in barycentric coordinates.
$P=(1-u-v)A+uB+vC=A+u(B-A)+v(C-A)$
Then if the ray intersects this triangle and hits P, then P is on the ray.
$P=O+tD$(O is the origin,D is direction)
From the two equations above, we can get the following equation:
$[-D\ (B-A)\ (C-A)][t\ u\ v]^T=O-A$
For every triangle, we already know its three vertices'position. So using the above equation, we get the **t** value, where O+tD is the intersecting hit point of the ray and the triangle.
Then update the surface's normal, primitive point, bsdf point at the hit point. Also remember to update **t_max** value of the ray to **t** value if **t<=current r.t_max** because minimum t guarantees the closest hit point of all triangles. This will be useful when we implement Bounding Volume Hierarchy method later.
The screenshots examples are as following:



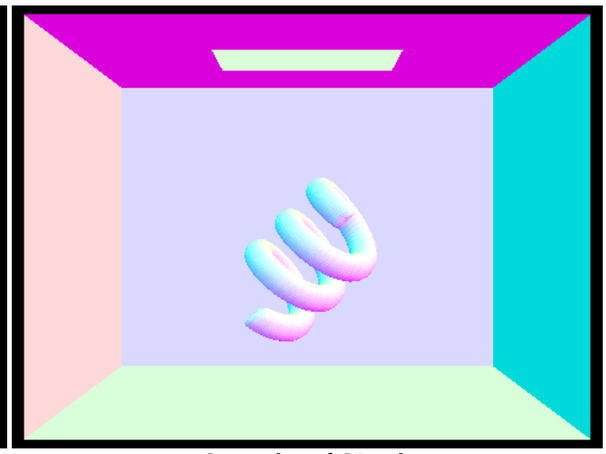
Screenshot of the dae/sky/CBspheres_lambertian.dae(no spheres appear this part)

Task 4:Intersecting Spheres

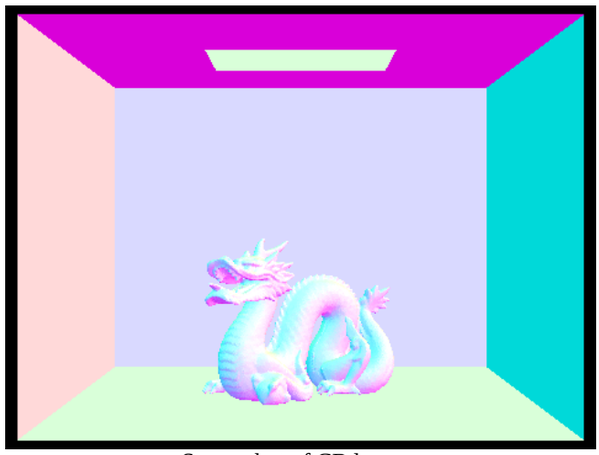
Similar to task 3, we have a ray and a sphere, the hit point P is on both the spere and the ray.
$(P-c)^2-R=0$
$P=O+tD$(O is the origin,D is direction)
From the above two equations, we can get **t** value reqresented by D,O,c. What differs from task 3 is we need to test whether the two solutions **t** of the equations can actually exist. Then we update the ray's corresponding values just like task 3.
The screenshots examples are as following:



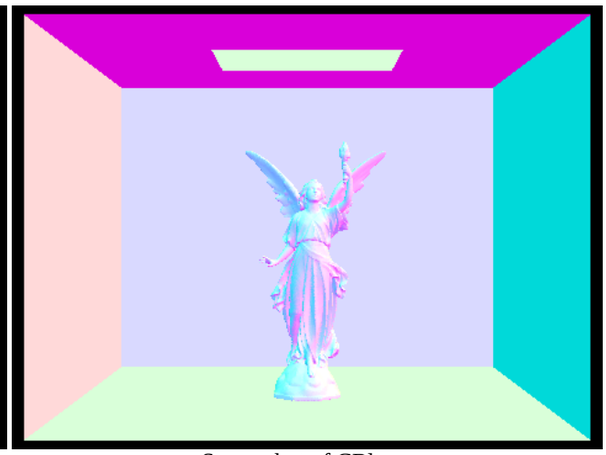
Screenshot of CBgems.



Screenshot of CBcoil.



Screenshot of CBdragon.



Screenshot of CBlucy.

After intersecting spheres, we can see the beautiful images above. However, when it gets more and more complicated, the time to render becomes very long because of the large amout of computation. For example, to render CBlucy.dae, it takes a few minutes. So we need to modify ray tracing to speed this process up. That is exactly what part 2 do.

Part 2: Bounding Volume Hierarchy

In part 1, we have implemented the ray generating and intersection and get the rendered images. Really amazing! But these delicate models always need a lot of time. What should we do? In this part, we use BVH to accelerate this process.

Task 1:Constructing the BVH

The main idea of bounding volume hierarchy is to devide the primitives into two bounding boxes and repeat this process until each bounding box has less than **max_leaf_size** primitives.
To construct the BVH, we first generate a cemtroid box which contains all primitives and calculate centroid of the large bounding box and every single primitive bounding box.
Then we need to decide which dimension to devide the primitives. One good way is to compare the dimension of the bounding box's extent and choose the largest one.
Next, we devide the primitives into two sets by comparing the centroid of every primite's bounding box and **centroid_box** on the **largest_dimension**.
Recurse, assigning the left and right children of this node to be two new calls to construct_bvh() with the two primitive lists just generated. Return the **node**.
However, when we choose the split point that makes all the primitives lie on one side of the split point(this may happen when a triangle has a vertex whose position is far away from other triangles), we will get a segfault because of infinite attempted recursive calls.
To deal with it, I use median object rather than the bounding box's midpoint to split the primitives when infinite recursive calls happen. More specifically, if the left child has nothing in it, I transfer half size of the right child to left child. Then we have finite recursive calls.

Task 2:Intersecting BBox

In this task, we use the simple ray and axis-aligned-plane intersection equation and the ray and axis-aligned-box intersection method to test the intersecting of the ray and the bounding box.
First using axis-aligned-plane intersection equation **t=(p-o)/d**, we get the range of t on corresponding axis(using min and max of bounding box on one axis).
Then using the ray and axis-aligned-box intersection method that the minimum t value is the maximum of the minimum of t on axis x,y,z, we get the **tmin tmax** value.
Comparing these two values to the actual **t** range of the ray, we can determine whether the ray intersects the bounding box.

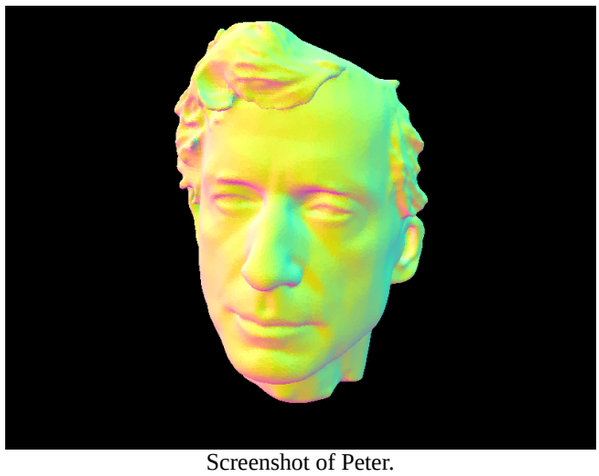
Task 3:Intersecting BVHAccel

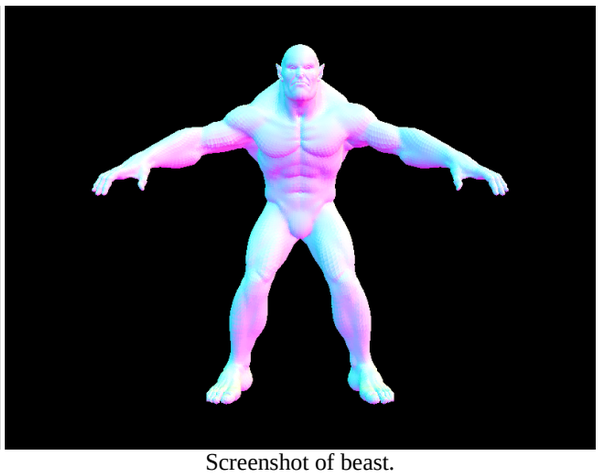
The main idea in this part is simple.
We now have the BVH structure from task 1 and we can test the intersection of the ray and bounding box from task 2. What we need to do is if the ray intersects the bounding box, recursively test if it intersects the left and right children of the bounding box. When reaching the leafnode which contains at most **max_leaf_size** primitives, test the intersection of the ray and the primitives, update the values of the ray. Because we always update the **t_max** value of the ray when test intersection, it is guaranteed that the ray stores the closer hit.

Here I make a mistake that **intersect(ray, i, node->l)||intersect(ray, i, node->r);**. But if the first part returns true, the second part will never be calculated. Since the function **intersect(ray, i)** updates the values of the ray, we must calculate the second part too. So I rewrite it as two parts a, b, and return **a||b**.
The screenshots examples are as following:


Screenshot of cow.


Screenshot of maxplanck.


Screenshot of Peter.


Screenshot of beast.

We can see that these models are quickly rendered and results are the same as using the original method which takes a really long time to render. To compare the efficiency of the original method and this modified one, I test their rendering time. For example, to render **cow.dae** without using BVH, it takes **73.5s**. With BVH modification, it takes only **0.0125s** to build BVH and **0.3365s** to render. It is much quicker than the original method in part 1!

Part 3: Direct Illumination

In this part, our goal is to get render images with realisitic shading. We devide this part into two tasks: first is to implement BSDF for simulating scattering. The second task is to implement direct lighting, whcih includes uniform hemisphere sampling and importance sampling by sampling over lights.

Task 1:Diffuse BSDF

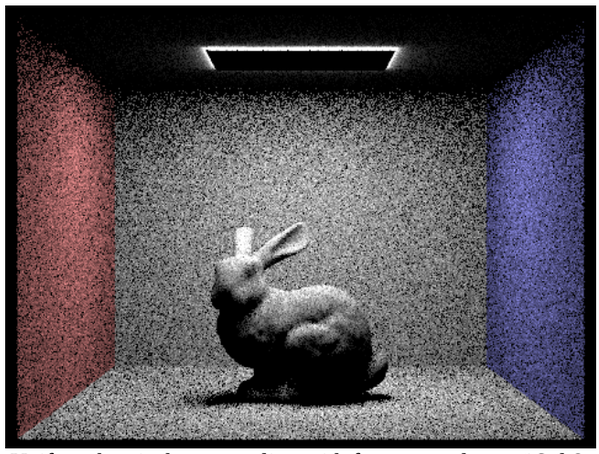
BSDF stands for bidirectional scattering distribution function. BSDF is mainly used to describe the way in which the light is scattered by a surface. Similarly, we have SRDF which describe how light is reflected at a surface. In this task, we are going to calculate BSDF.
Since light is equally reflected in each output direction, we calculate BSDF **f=reflectance/PI**. The reflectance here is also known as albedo. The albedo is a measure of how reflective the surface is, ranging from 0 (total absorption) to 1 (total reflection). As for function **sample_f**, we first get a sample using **sampler.get_sample**. Then get the BSDF from this sample.

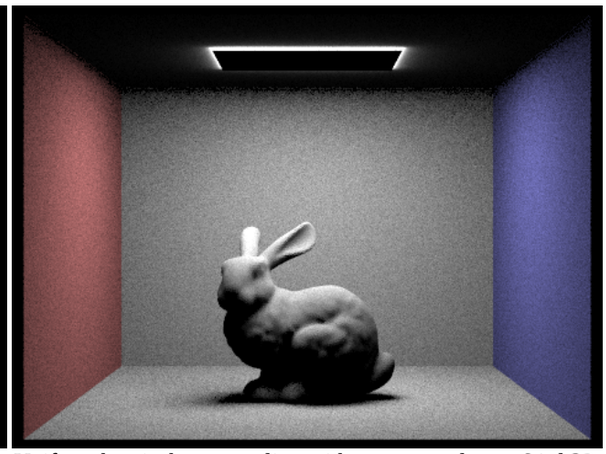
Task 2: Direct lighting

To get an estimate of the direct lighting at a point that a ray hits, we have two methods: uniform hemisphere sampling and importance sampling by sampling over lights.

Uniform hemisphere sampling

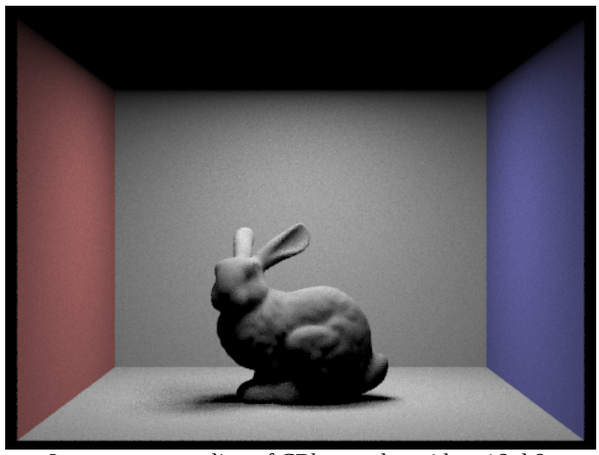
Uniform hemisphere sampling is a method that takes sample points in a uniform hemisphere around the point of interest, and for each ray that intersects a light source, compute the incoming radiance from that light source, then convert it to the outgoing radiance using the BSDF at the surface, and finally average across all samples to get the spectrum.
When we take more samples points, which means more rays are taken, it will give a smoother image since more rays will hit the light source.
Here are some scheenshot examples of uniform hemisphere sampling.

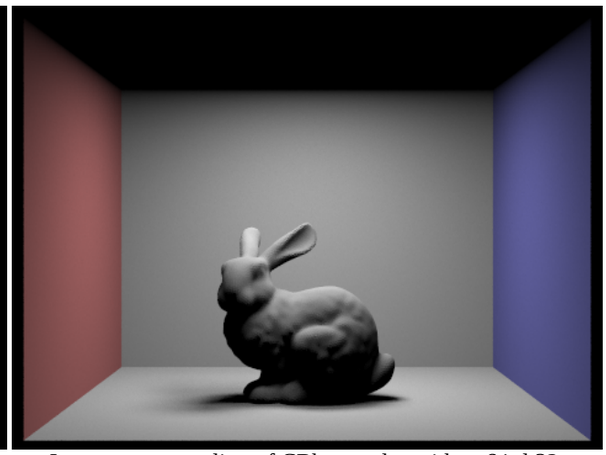

Uniform hemisphere sampling with fewer samples, -s 16 -l 8.


Uniform hemisphere sampling with more samples, -s 64 -l 32.

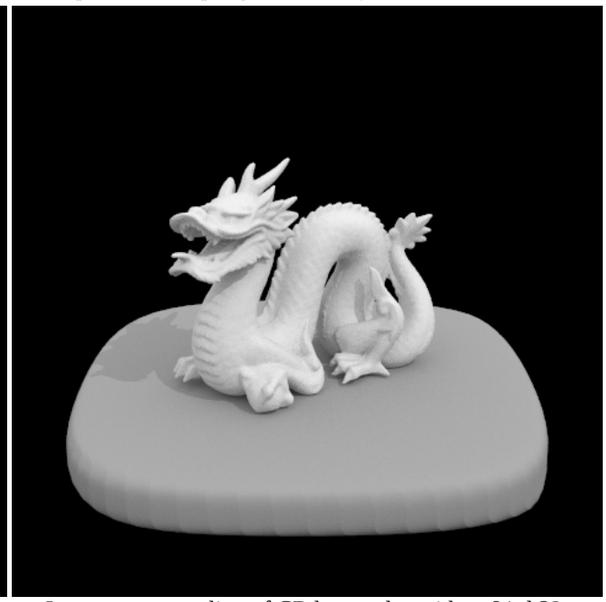
Importance sampling

Because there exists many black pixels using uniform hemisphere sampling (since many sample rays don't intersect light source), we change the basis of integration to sample light source area. Instead of uniformly sampling in a hemisphere, we sample each light directly. Thus, we can cast rays that intersect light source area.
More specifically, we sum over every light source in the scene(light source area), taking samples from the surface of each light, computing the incoming radiance from those sampled directions, then converting those to outgoing radiance using the BSDF at the surface. Here are some scheenshot examples of importance sampling.

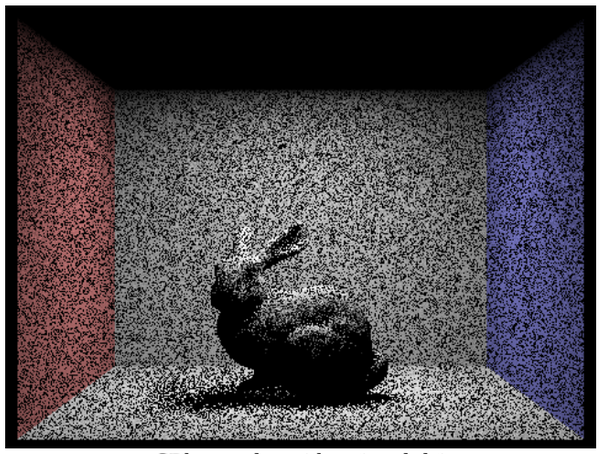

Importance sampling of CBbunny.dae with -s 16 -l 8.


Importance sampling of CBbunny.dae with -s 64 -l 32.

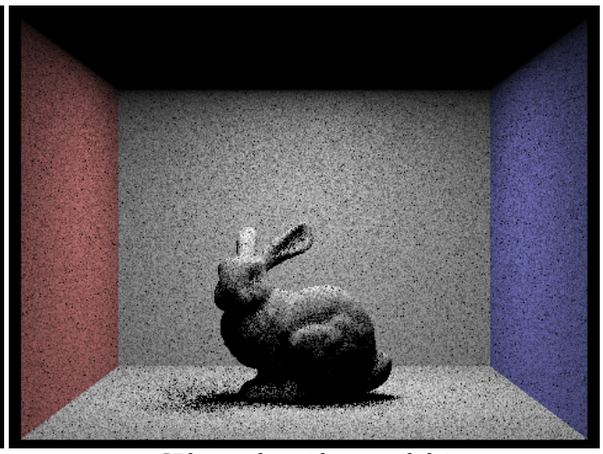

Importance sampling of CBdragon.dae with -s 16 -l 8.


Importance sampling of CBdragon.dae with -s 64 -l 32.

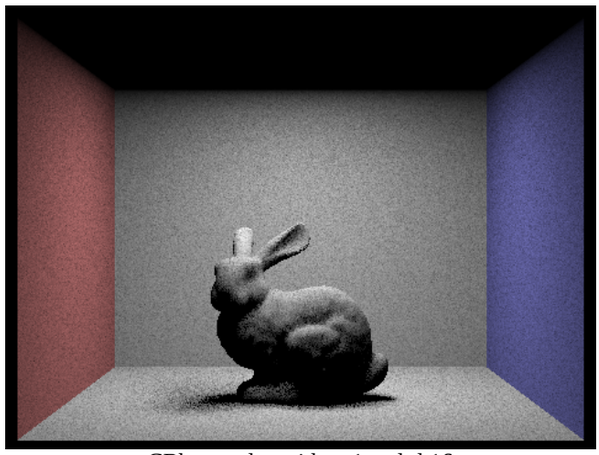
To test the noise levels of different number of light rays, I render with 1,4,16,64 light rays and 1 sample per pixel using importance sampling. The screenshots are as following:



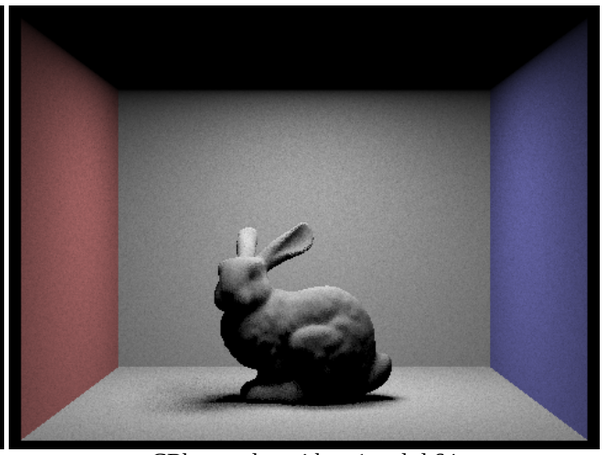
CBbunny.dae with -s 1 and -l 1



CBbunny.dae with -s 1 and -l 4



CBbunny.dae with -s 1 and -l 16

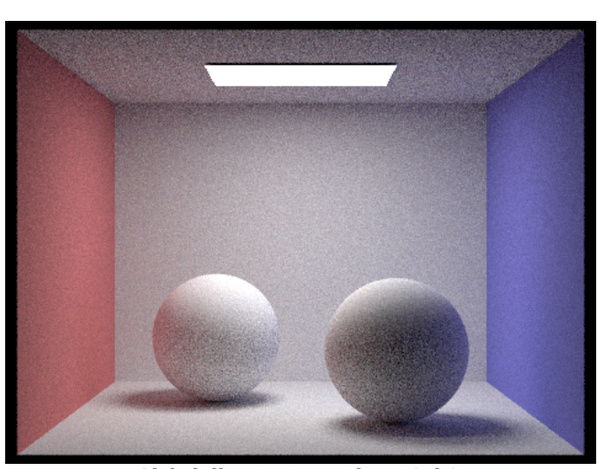


CBbunny.dae with -s 1 and -l 64

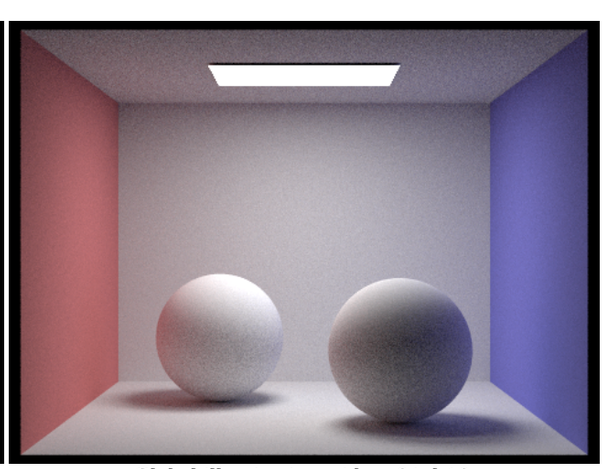
We can find that importance sampling by sampling over lights can give a less noisy rendered image since when the number of rays is the same, it is better than uniform hemisphere sampling. The reason why it is less noisy is that importance sampling either return the spectrum of the light source or hit an obstacle object. As for uniform hemisphere sampling, since we randomly cast rays through the point, we may return black if the rays happen not to intersect the light source. That results in the appearance of more black pixels using hemisphere sampling. Plus, importance sampling is less affected by the number of light rays compared with uniform hemisphere sampling because it at least directs all rays towards a light, whereas hemisphere sampling chooses directions totally randomly. So we can see that importance sampling is better than uniform hemisphere sampling.

Part 4:Global Illumination

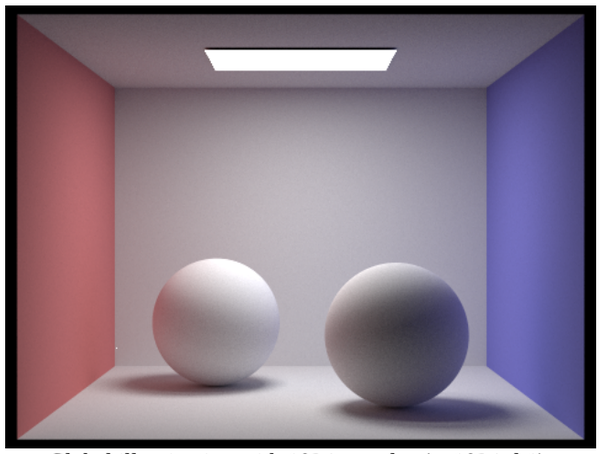
In last part, we get the direct lighting of every pixel. But since there exists more than one bounce reflection, if we want a more realistic rendered result, we must include not only one bounce radiance, but also the emission that from itself and the radiance that generated from more than one bounce. Since we have already implemented the function **one_bounce_radiance** and function **zero_bounce_radiance** simply returns **get_emission()**, the main task is to implement function **at_least_one_bounce_radiance**.
The function **at_least_one_bounce_radiance** should call **the one_bounce_radiance** function as the base case, and then recursively call itself to estimate the higher bounces. This recursive call should take one random sample from the BSDF at the hit point, trace a ray in that sample direction, and recursively call itself on the new hit point.
To terminate bouncing, we use Russian roulette and set the maximum of the ray depth. If the Russian roulette decides we should continue or we haven't reach the maximum ray depth, we continue calling **at_least_one_bounce_radiance**. Accumulate every bounce. Then we add **zero_bounce_radiance** and **at_least_one_bounce_radiance** to get the global illumination.
Here are some scheenshot examples of global illumination.



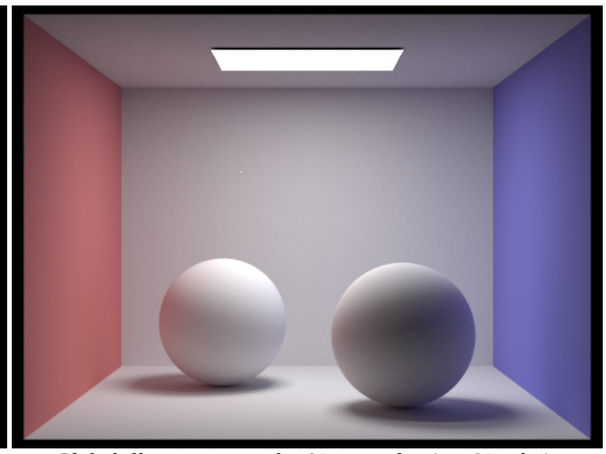
Global illumination with -s 16 -l 8.



Global illumination with -s 64 -l 16.

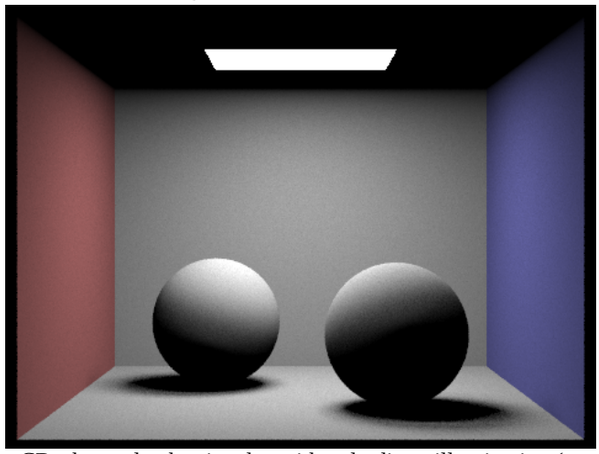


Global illumination with 1024 samples (-s 1024 -l 1).

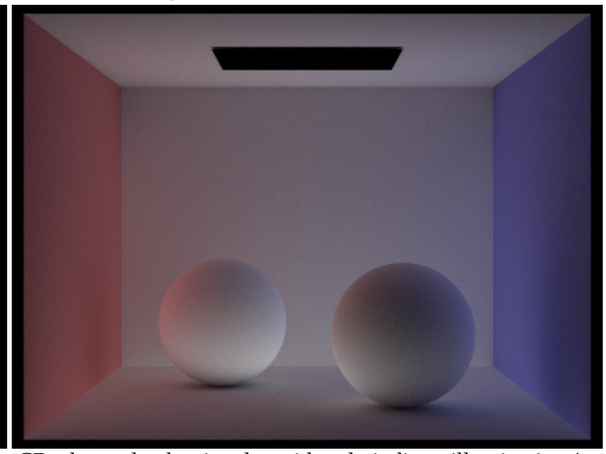


Global illumination with 1024 samples (-s 1024 -l 4).

To figure out what exactly direct illumination and indirect illumination are like, I render with only direct illumination and only indirect illumination. The rendered views are the following.

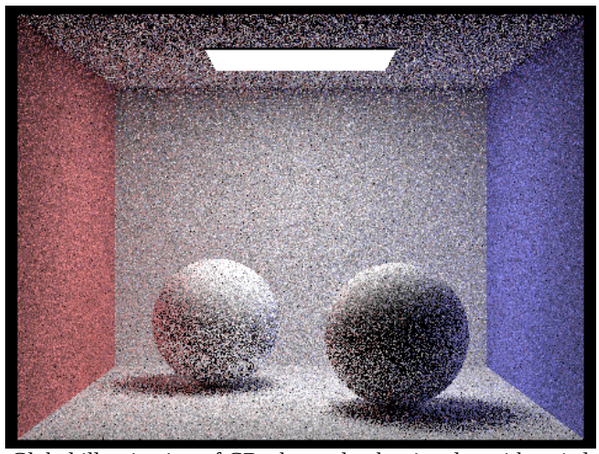


CBspheres_lambertian.dae with only direct illumination (-s 16 -l 16).

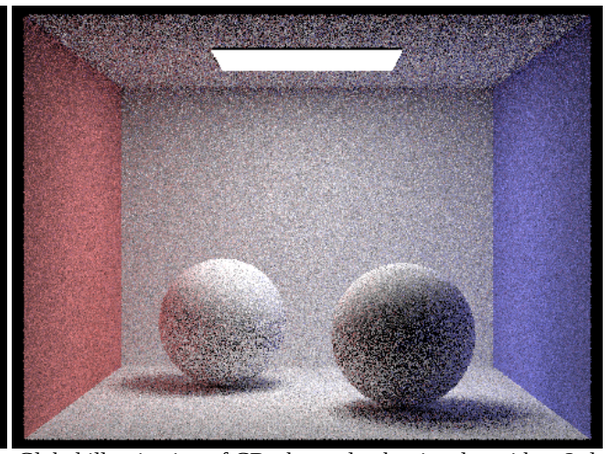


CBspheres_lambertian.dae with only indirect illumination (-s 16 -l 16).

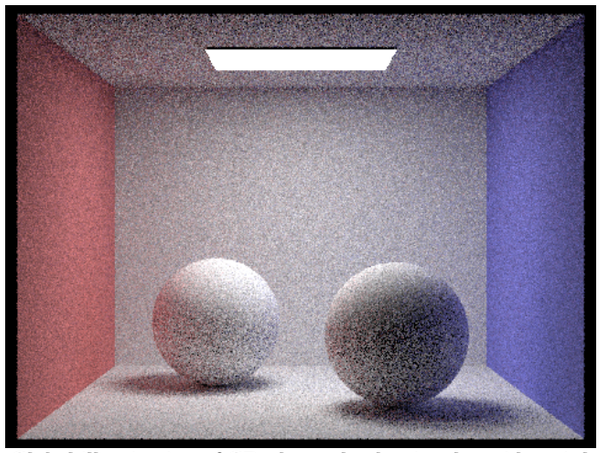
Then we pick one scene and compare rendered views with various sample-per-pixel rates, including 1, 2, 4, 8, 16, 32, 64, and 1024, using 4 light rays.



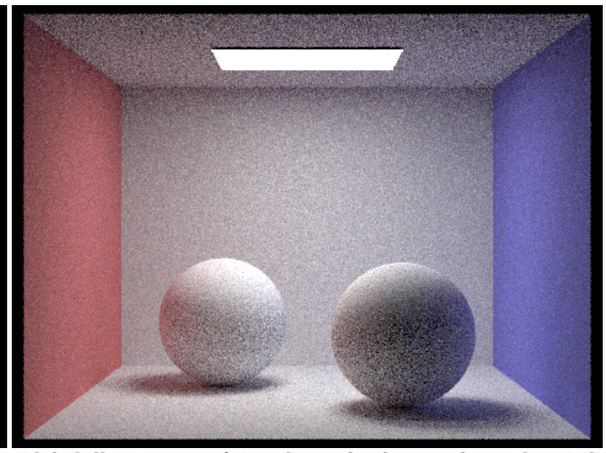
Global illumination of CBspheres_lambertian.dae with -s 1 -l 4.



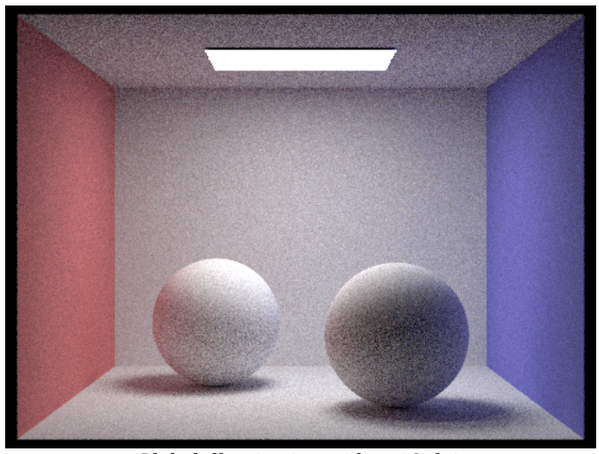
Global illumination of CBspheres_lambertian.dae with -s 2 -l 4.



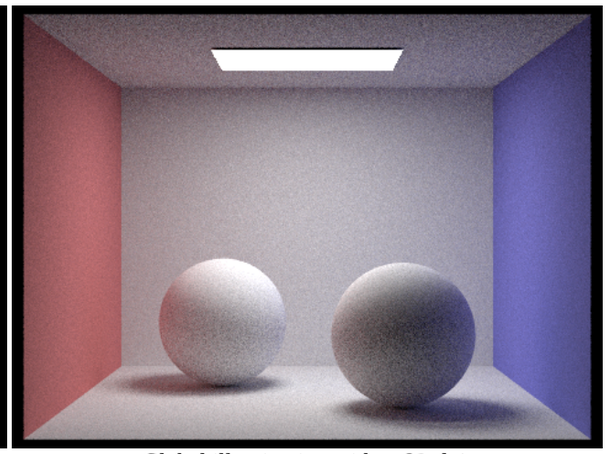
Global illumination of CBspheres_lambertian.dae with -s 4 -l 4.



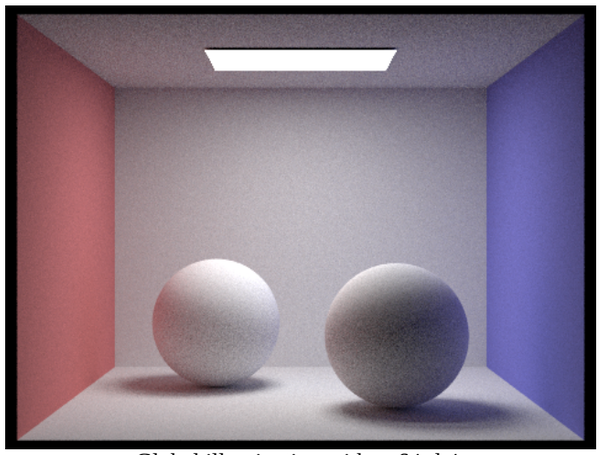
Global illumination of CBspheres_lambertian.dae with -s 8 -l 4.



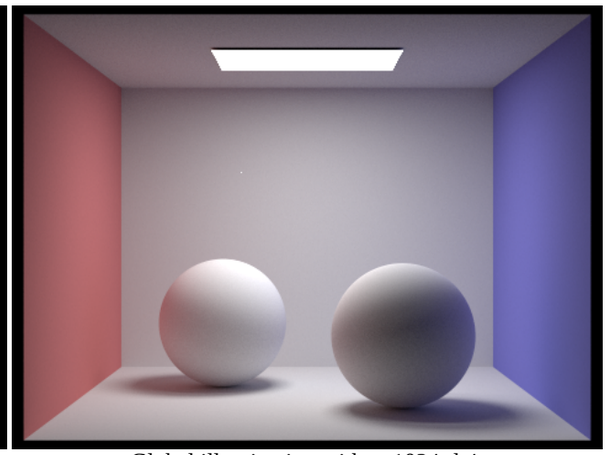
Global illumination with -s 16 -l 4.



Global illumination with -s 32 -l 4.



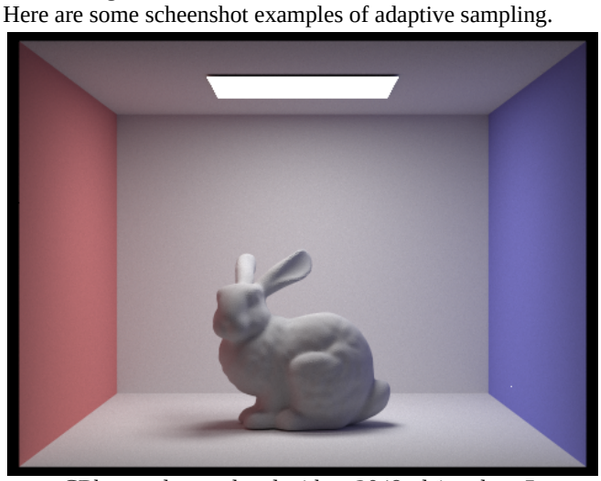
Global illumination with -s 64 -l 4.



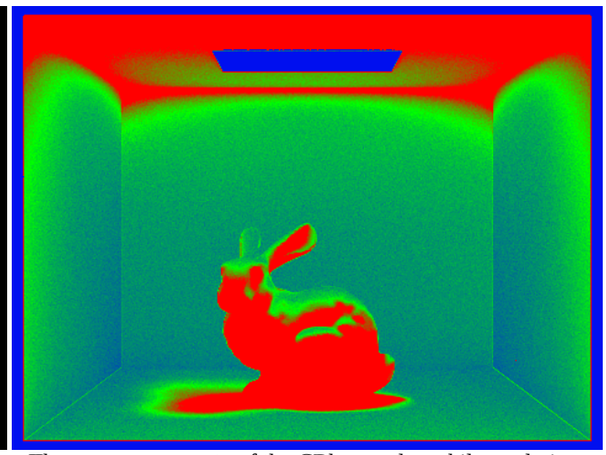
Global illumination with -s 1024 -l 4.

Part 5: Adaptive Sampling

Although we already have some amazingly rendered images, there still exists noise. In this part, we want to completely eliminate noise without having to uniformly raise the number of samples per pixel.
In a model, some pixels converge faster with low sampling rates, while other pixels require many more samples to get rid of noise. So we use a method called adaptive sampling, which means we concentrate the samples in the more difficult parts of the image. It works for each pixel individually based on statistics.
Define a variable **I=1.96*standard deviation/mean**. And if **I** is less than max_tolerance multiplies by mean, we can say that with 95% confidence, the average illuminance in this pixel is between mean-I and mean+I, based on our nn samples so far. To calculate the mean and standard deviation, we set a batch and calculate them through the batch.
Here are some scheenshot examples of adaptive sampling.



CBbunny.dae rendered with -s 2048, -l 1 and -m 5.



The convergence rate of the CBbunny.dae while rendering.