Assignment 3: PathTracer

Jacob Hsiung

This project is completely about pathtracing just like its name. We start off with something simple. To begin with, we wrote a simple ray-generating algorithim so we can do ray tracing from there. Secondly, we implemented a intersection test, including one for traingle and another for sphere. At this point, we are able to do the simplest ray tracing. However, the problem was that, it took way too long to render just a low resolution image. This took us to the next part of the project which is the BVH Acceleration algorithim. We partition the space into several bounding boxes, and each bounding boxes will contain several primitives. This structure speeds up the ray tracing algorithim significantly. With the speed-up version of ray tracing, we can start on direct illumination and then global illumination. The first part of direct illumination is hemisphere sampling and importance sampling. The former sample the entire hemisphere at uniform distribution, so we end up sampling lots of non-light sourve object, causing the image to be noisy . This problem can be solved by the latter algorithim which is importance sampling. This algorithim directly loop through all the light source in the sapce and sample its radiance. This guarantees that all of our sample are on the "important" source. We can achieve higher quality image with the same computational cost. After completing direct illumination, we can further extend our path tracing algorithim by enabling global illumination. In direct illumination, we are only considering radiance directly coming from light source, what we ignored is the raidance being reflected off the object. To include those radiance, we have to take the BRDF of different object into consideration. By using a recursive algorithim and Russian Roulette we can recursively calculate the radiance being reflected around. The details of the implementation be further discussed later. Last but not least, to make our sampling more efficient, we applied the method called adaptive sampling, which essentially stops sampling a pixel when its pixel value has converged.

Part 1: Ray Generation and Intersection

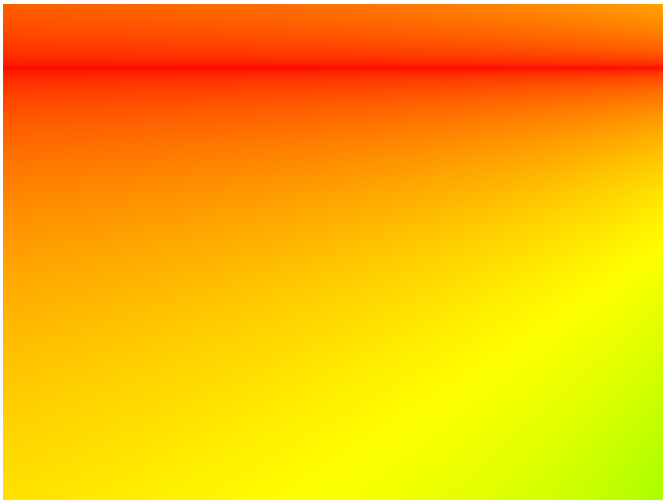


image with normal shading

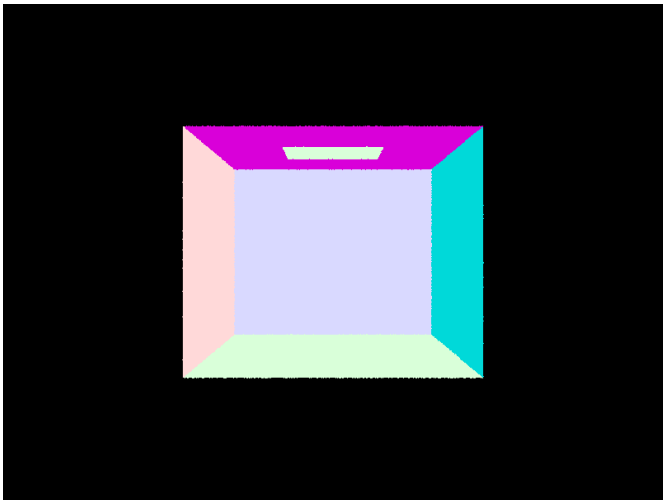


image for intersection

The ray generation algorithim was rather straighfoward. To generate a ray in the right direction, we first generate the ray using the fact that xmax is tan(hFov) and ymax is tan(vFov) in the image plane. Using these two values, we can interpolate the coordinates at the given x, y value, and the z coordinates are always set to -1. At this point, we can calulate the direction of this ray since the origin of the ray concides with the origin of the coordinates. Furthermore, we have to normalize the direction of the ray and setting its min_t to nClip and max_t = fClip. The importance of setting these value will become obvious in the latter part of the project because it helps us determine whether or not the intersection between a ray and an object is valid. Lastly, the generated ray is in the camera coordinates, so we need to convert it into the worlds coordinates, which can be dome by multiplying with the c2w matrix. The traingle intersectin algorithm I implemented was Moller Trumbore algorithm. According to the algorithm, the intersection point is given by O + tD = (1-b1-b1)*P0 + b1*p1 + b2* p2. P0, P1, P2 are the three vertex of the triangel and b1 and b2 are variables that we will calculate. We first define a few constants that are E1 = P1 - P0, E2 = P2 - P0, S = O -P0, S1 = cross(D, E2), S2 = cross(S, E1). With these constants, [t, b1, b2].T = 1/(cross(S1, E1)) * [dot(S2, E2), dot(S1, S), dot(S2, D)].T. We then check if t is in the range of min_t and max_t of the ray, b1 and b1 are greater than 0, b1 + b2 are samller than 1, and t is greater than 0. If all of the above conditions are satisfied, then the intersection point is said to be within the traingle.

Part 2: Bouding Volume Hierarchy

To construct the BVH Heirarchy. Firstly, I loop through all the primitives and add all of them to the root bounding box. I also added all of their centroid to a ceontroid bounding box, which is later used for splitting. If the total number of primitives in the bounding box is greater than the max_leat_size allowed, then we have to split the bounding box into left and right node. I chose the axis with the greatest extent according to the centroid bounding box we just calculated. I split along the mid point of the axis, and seperate all of the primitiveses into left and righ vectors based on their centroid. After seperation, I recursively called construct_bvh on the left and right node of the bounding box with the vector l and r. The recursion will end when the number of primitives in the bounding box is less then the max_leat_size.

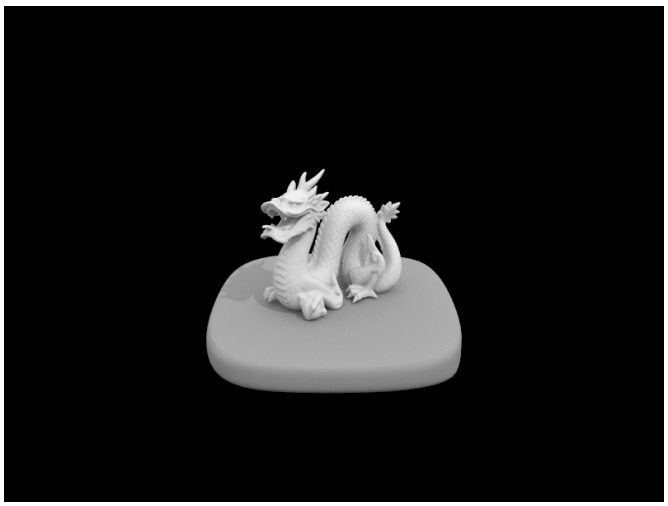
cow.png with BVH acceleration

MaxPlank.png with BVH acceleration

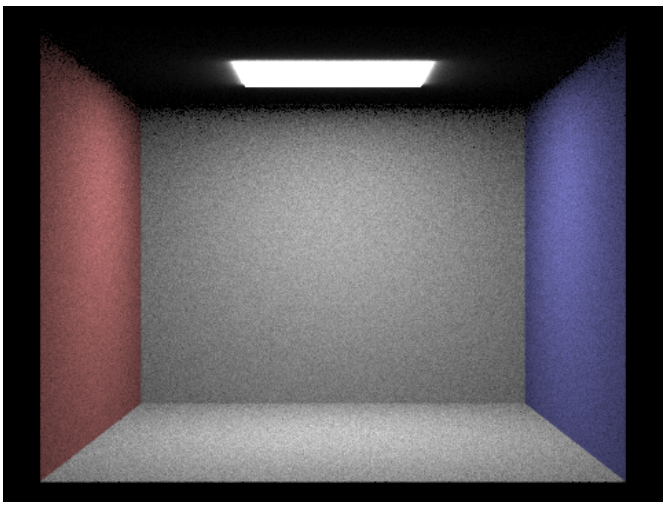
Before the BVH, cow.png takes about 62 seconds to render, but then is reduced to less than a secdnd. The MW.png orginally takes 239 second to render, but it was reduced to also less than a second. The reason behind the incredible speed up is because we decrease the number of intersection between the ray and objects. Instead of checking intersection primitive by primitive, we only check for intersection when the ray intersect the bounding box in the first place.

Part 3: Direct illumination

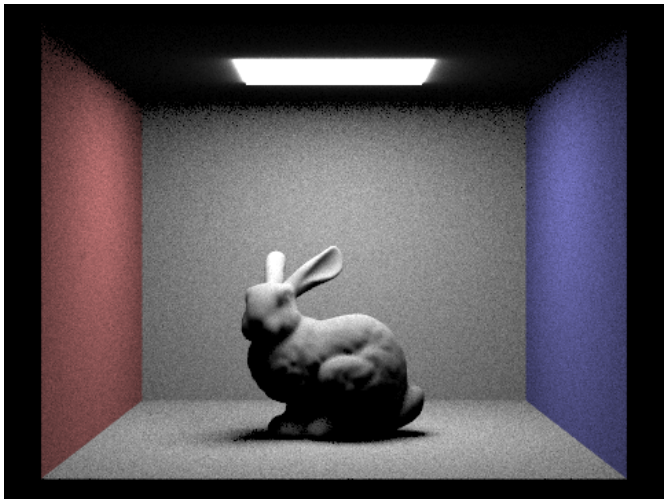
In the first half of this part, we implemented the hemisphere sampling. In this method, we do not know where the light source is located in the hemisphere, so we radomly sample the ray in the hemisphere. It follows that the chance of hitting a light source is small, not to mention if it is a point light source. However, if our sample ray do intersect with a light source, we calculate how much radiance is emitted from the light source. If the sample ray do not intersect with a light source, we simply move on to the next sample. At the end, we have to normalize our sample by multiplying it with 2 PI and divided by the total number of sample taken. In the second half, we implemented the importance sampling. This method addresses the problem of hemisphere sampling, which is that lots of our sample ray doesn't intersect with the light source; hence, we cannot produce images with stable and high quality. To solve the problem, instead of randomly sample the ray, we iterate over all the light sources, then sample a ray from the light source to the object. We then have to verify that this ray doesn't intersect anythingwhen traveling from the light source to the object, so that the radiance emitted from the light source is directly onto the object. In the end, we get the bsdf of the object and multiply it with the incoming radiance from the light source and multiply it with the cos theta of the radiance and normalize it by dividing by the pdf sample distribution. There are two things to notice in importance sampling. First thing is that if the light source is a point light source, then we can reduce the number of sample to 1. Second thing is that when calculting the radiance, we have to take into consideration that this is radiance is sampled with a probability pdf from the light sourve, so we have to normalize it by divding pdf.



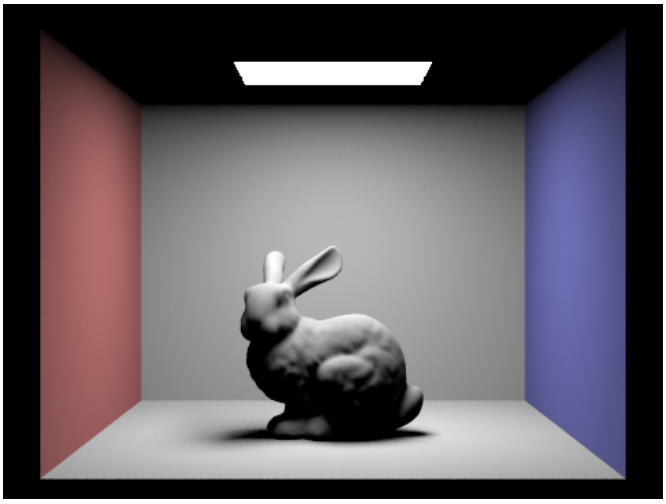
Importance sampling 64 per pixel 32 per pight



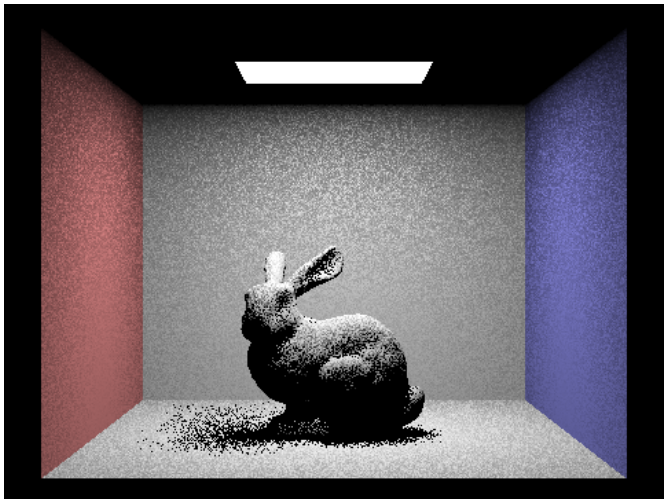
Importance sampling 64 per pixel 32 per pight



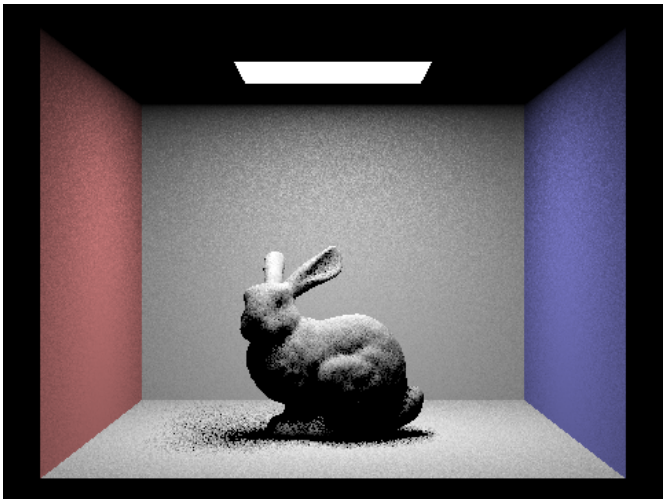
Hemisphere sampling 64 per pixel 32 per pight



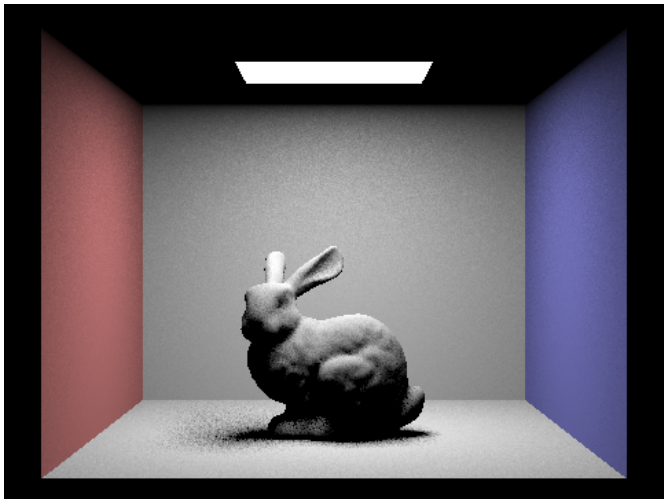
Importance sampling 64 per pixel 32 per pight



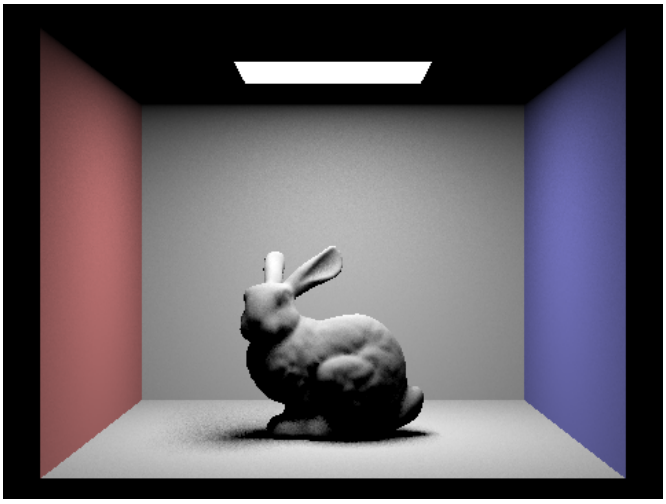
Importance sampling 1 per pixel 1 per pight



Importance sampling 1 per pixel 4 per pight

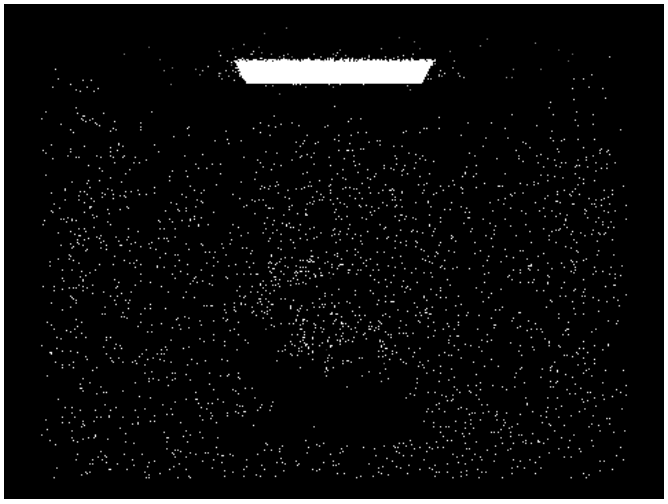


Importance sampling 1 per pixel 16 per pight

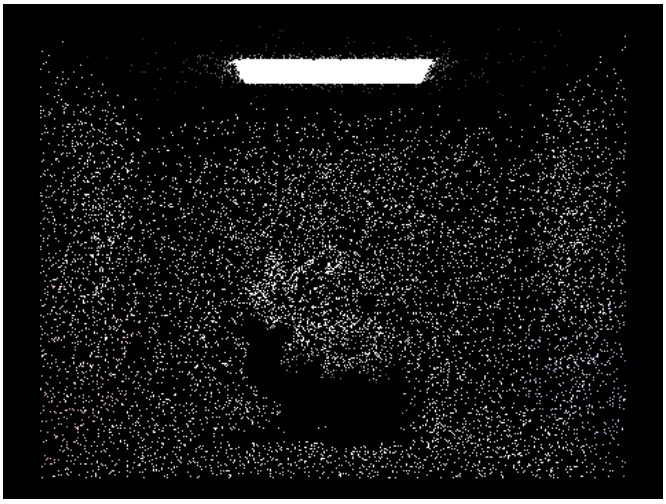


Importance sampling 1 per pixel 64 per pight

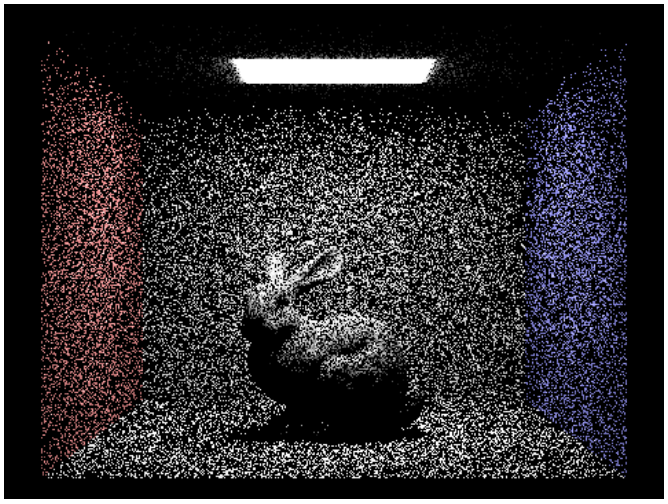
As we increase the number of samples for light, we can see that the noises decreases dramtically even though the sample per pixel is set to 1



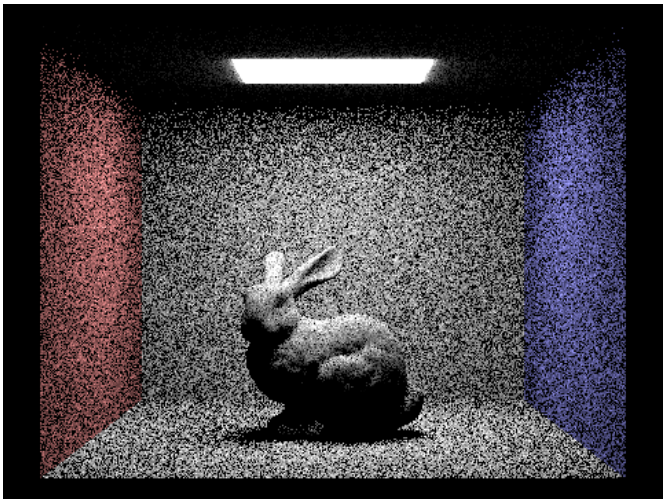
Hemisphere sampling 1 per pixel 1 per pight



Hemisphere sampling 1 per pixel 4 per pight



Hemisphere sampling 1 per pixel 16 per pight

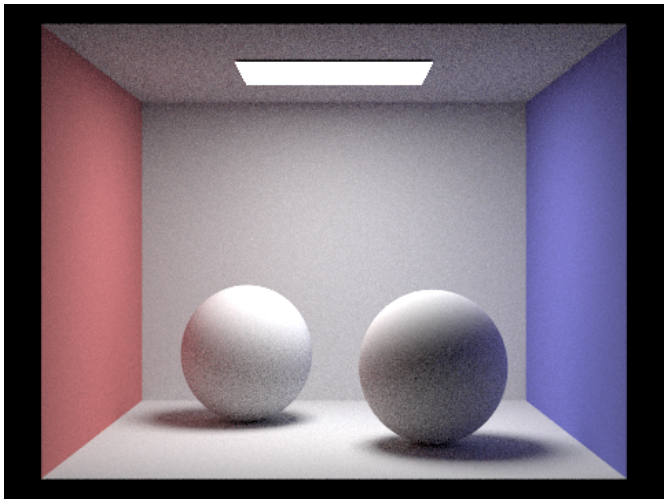


Hemisphere sampling 1 per pixel 64 per pight

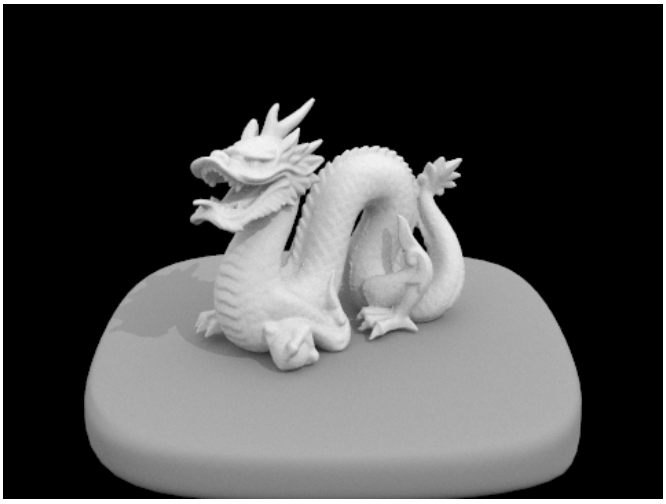
Comparing the result of Hemimsphere sampling and Importance sampling, we can see that with the same parameters, importance sampling can achieve better quality. This is very obvious if we are sampling 1 ray per pixel. In Hemisphere sampling, we have a big chance of missing the light sourve resulting in a black pixel, causing lots of noise in our image. However, in Importance sampling, we will be sampling in the light source direction.

Part 4: Global illumination

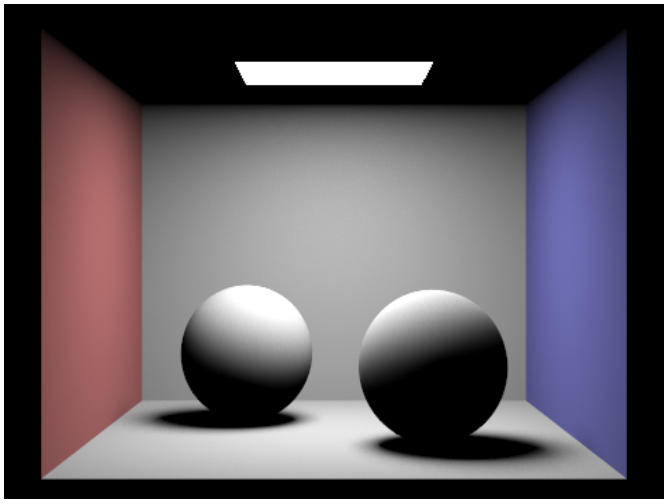
We already implemented zero bounce and one bounce radiance in part 3 of this project. However, there are still lots of noice in our image. To produce better quality image, we have to consider radiance that bounces more than 1 time. We implement the at_least_one_bounce function which calles one_bounce_function in the beginning and recursively calls itself. In each recursive call to the functnion, it sample a new direction from the bsdf of the object and generate a ray in that direction. If the ray intersects with a object, it recursively call itself with the new isect object and the ray that just intersect with it. To prevent infinite recursion, we use the Russian Roulette method, which terminates the recursion with a probability p. This p is chosen by the user. Besides Russian Roulette. every ray has a member variable depth. In each recursion, we decrease the value of depth by 1, when the depth is 0, we also terminates the recursion. In the end of the algorithim, we add together zero_bounce and at_least_one_bounce to get the estimate of global illumination.



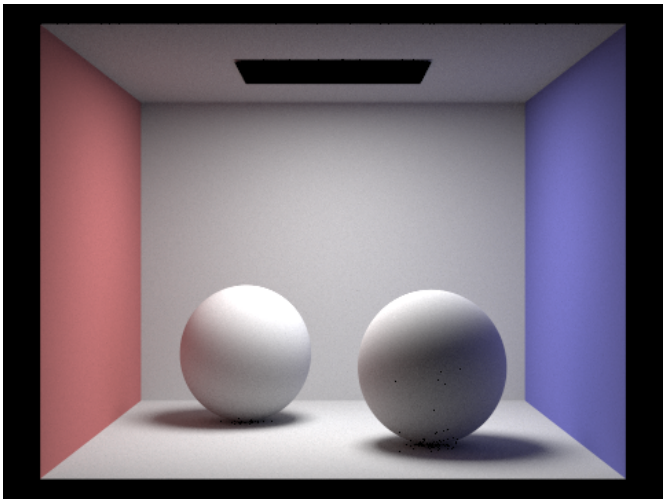
Global Illumination of Spheres



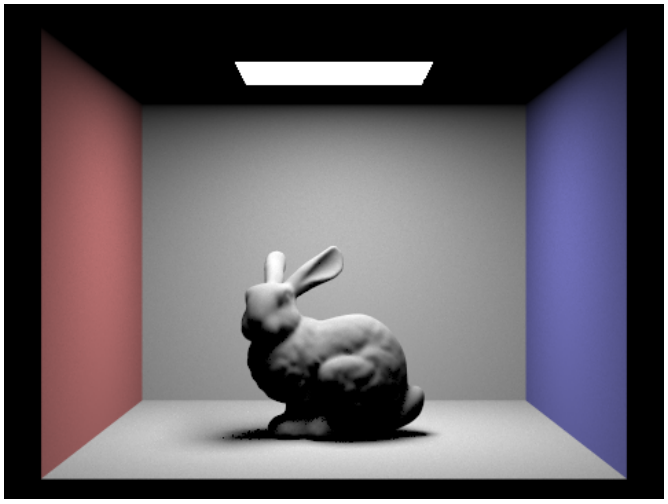
Global Illumination of Dragon



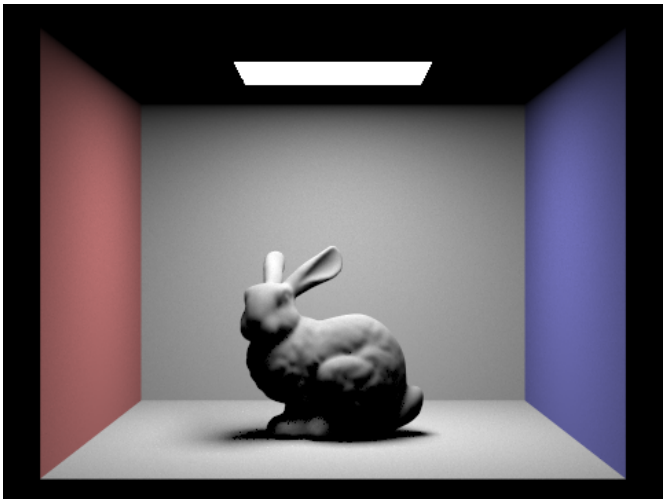
Direct illumination



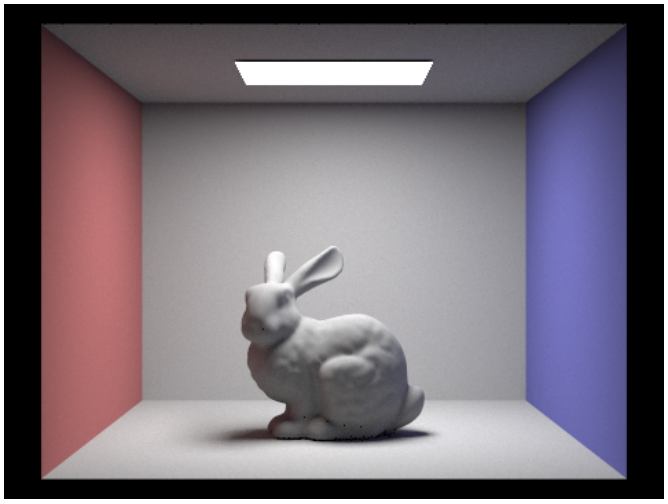
indirect illumination



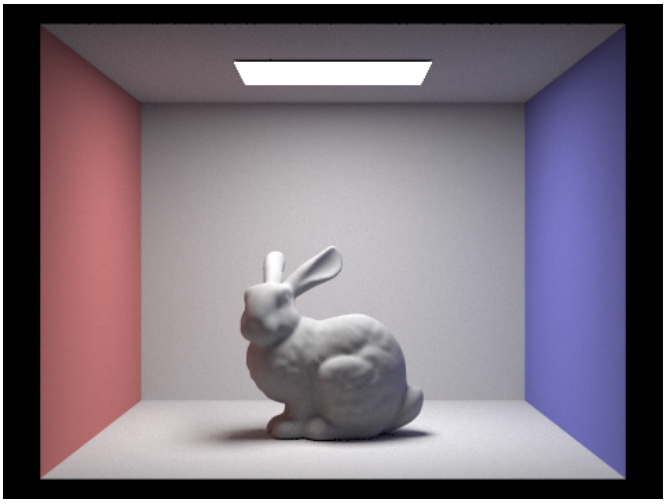
1024 samples per pixel, 1 sample per light, Depth = 0



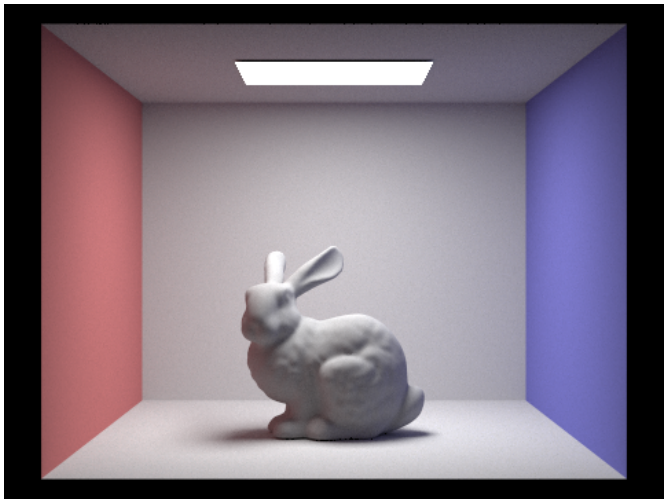
1024 samples per pixel, 1 sample per light, Depth = 1



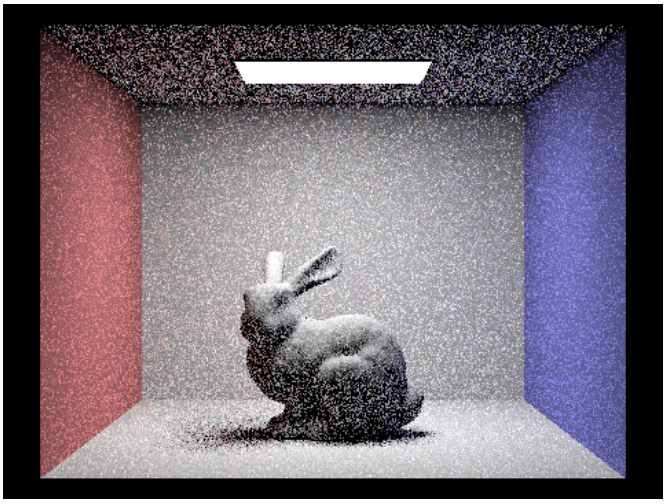
1024 samples per pixel, 1 sample per light, Depth = 2



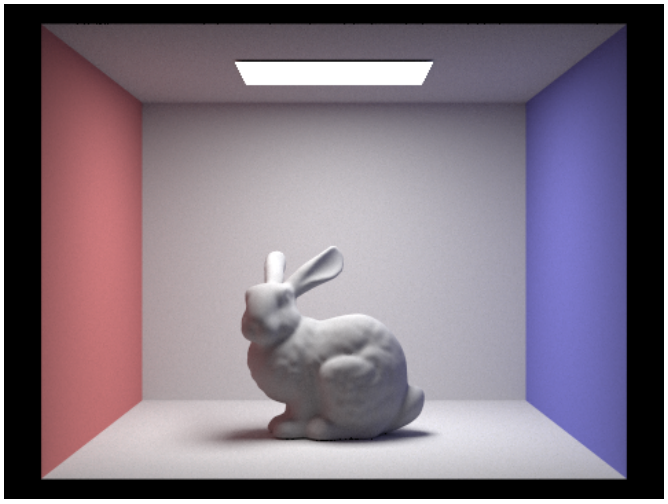
1024 samples per pixel, 1 sample per light, Depth = 3



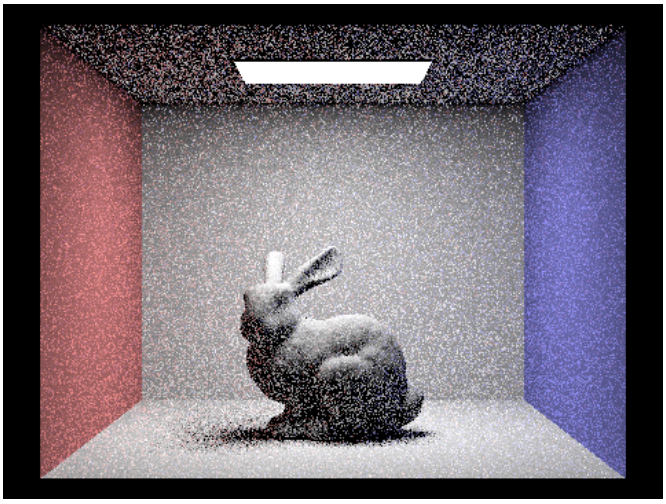
1024 samples per pixel, 1 sample per light, Depth = 100



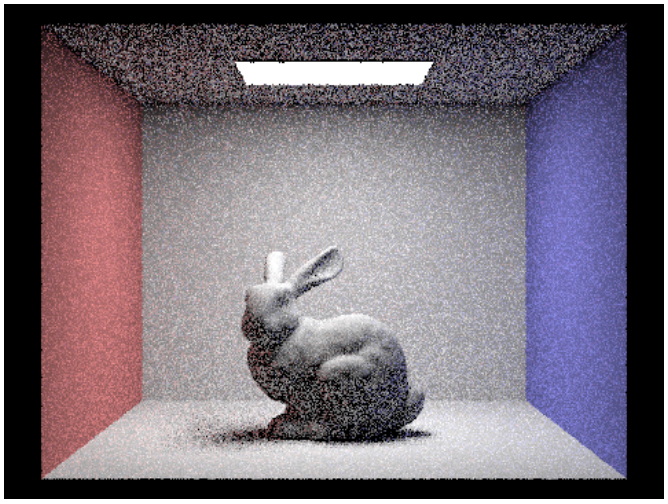
1024 samples per pixel, 1 sample per light, Depth = 100



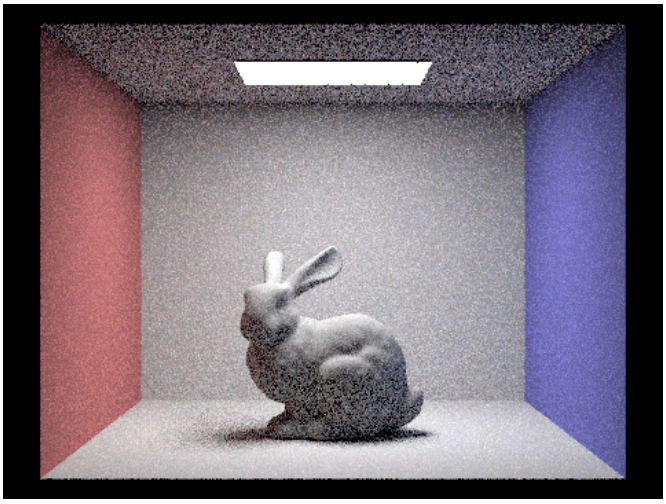
1024 samples per pixel, 1 sample per light, Depth = 100



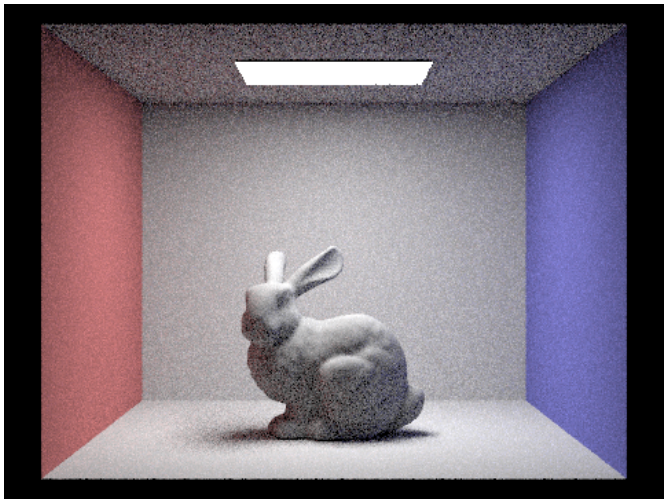
1 samples per pixel, 4 sample per light, Depth = 3



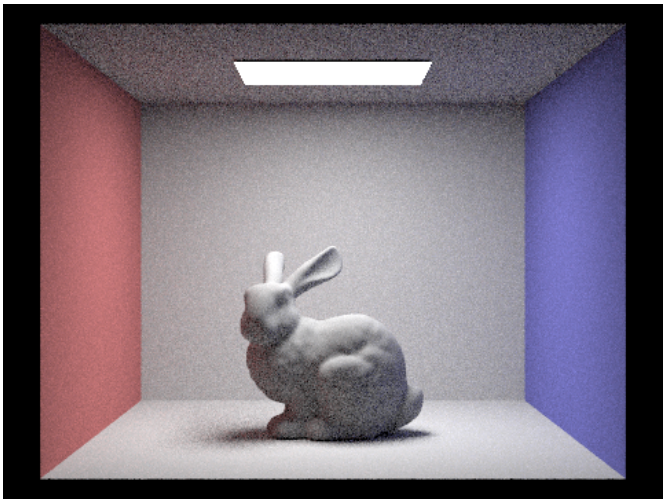
2 samples per pixel, 4 sample per light, Depth = 3



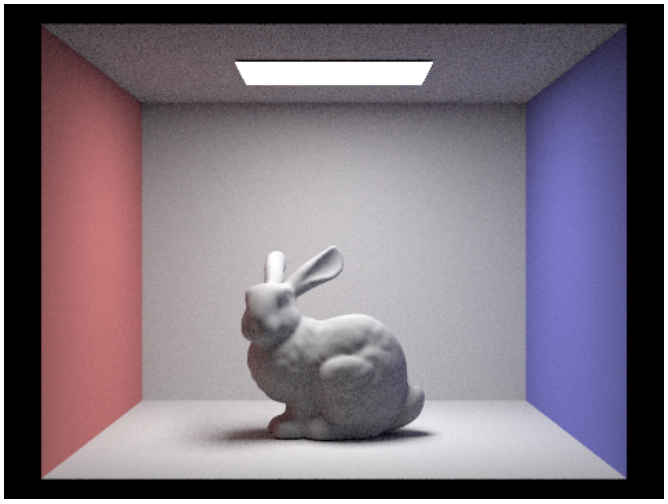
4 samples per pixel, 4 sample per light, Depth = 3



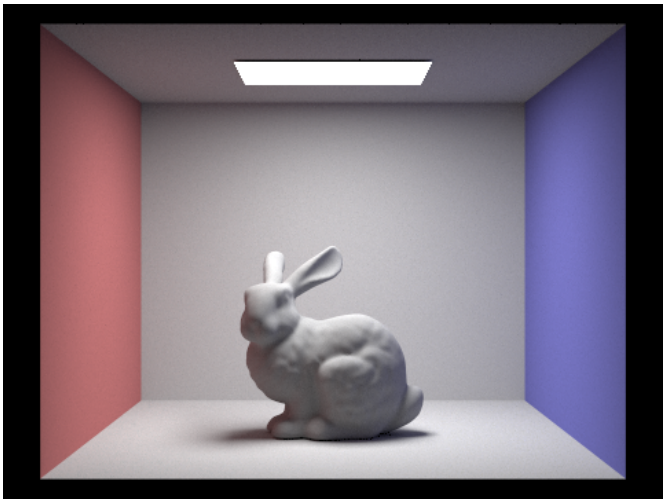
8 samples per pixel, 4 sample per light, Depth = 3



16 samples per pixel, 4 sample per light, Depth = 3



64 samples per pixel, 4 sample per light, Depth = 3



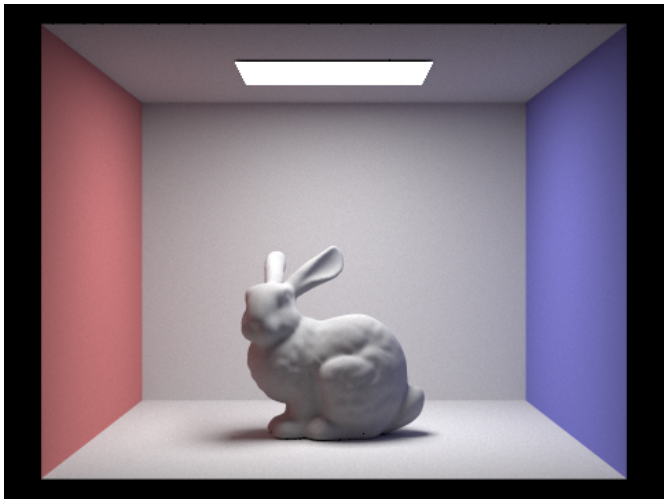
1024 samples per pixel, 4 sample per light, Depth = 3

For CBbunny.dae, when we set the number of samples to 1024 per pixels and render image with max_ray_depth = 0,1,2,3,100. As we increase the max_ray_depth, the image become brighter and brighter because they are brighten by the gloabel illumination.

On the other hand, when we set the samples per light to 4 and sample per pixel = 1,2,4,8,16,64,1024. The noise in our image become less and less. The image become clearer as we increase the sample per pixel.

Part 5: Adaptive Sampling

Adaptive concept is a relatively straightfoward concept. In the past, we sample every pixel a set amount of times no matter what the value is. In adaptive sampling, however, we stop sampling the pixel if the value of the pixel converge. We can check if the value of the pixel converges by calculating its I, which is equal to 1.96 * the standard deviation and divided by the square root of the number of samples taken so far. If I is less that the maxTolerance * mean; then we stop sampling the pixel.



2048 samples per pixel, 1 sample per light, Depth = 5

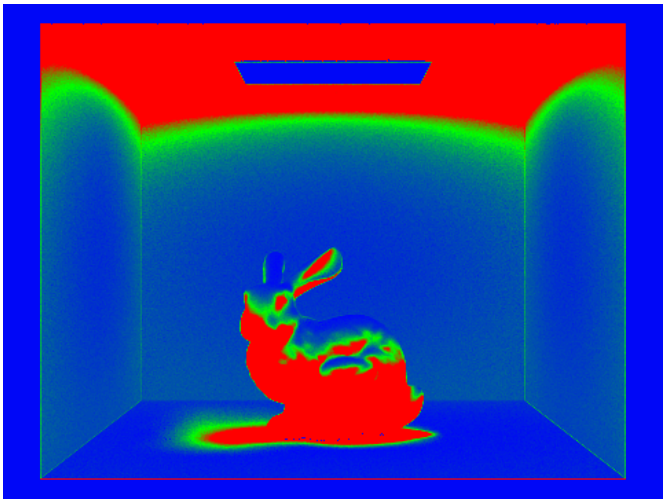


Image of sampling rate

From the image of sampling rate, we can see that, different from the uniform samling rate accross the image. We stop sampling the light source very quickly because the value of the pixel converges. However, notice that places with shadows often requires more samples per pixel before its value converges.