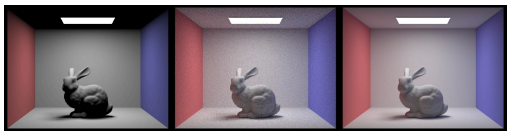
CS184/284A Spring 2025 Homework 3 Write-Up

Names: Andy Zhang

Link to webpage: https://github.com/cal-cs184-student/hw-webpages-zhangnd16
Link to GitHub repository: https://github.com/cal-cs184-student/sp25-hw3-escher



Overview

In this project, I have implemented several important concepts of ray tracing, which is very important in rendering 3D scenes. I implemented optics formula into programs to calculate the time and intersection of a ray intersecting a triangle. Then, to accelerate this process, I implement BVH tree so that the program does not have to traverse every primitives. Then, I implemented direct illumination and indirect illumination, to simulate the realistic lighting coloring effect of a 3D object in the scene. Lastly, I implemented a simple adaptive sampling algorithm to accelerate this process. I found that optics are very helpful in simulating realistic lighting, and simple data structures, such as BVH trees, can help to drastically improve the performance of algorithms in computer graphics.

Part 1: Ray Generation and Scene Intersection

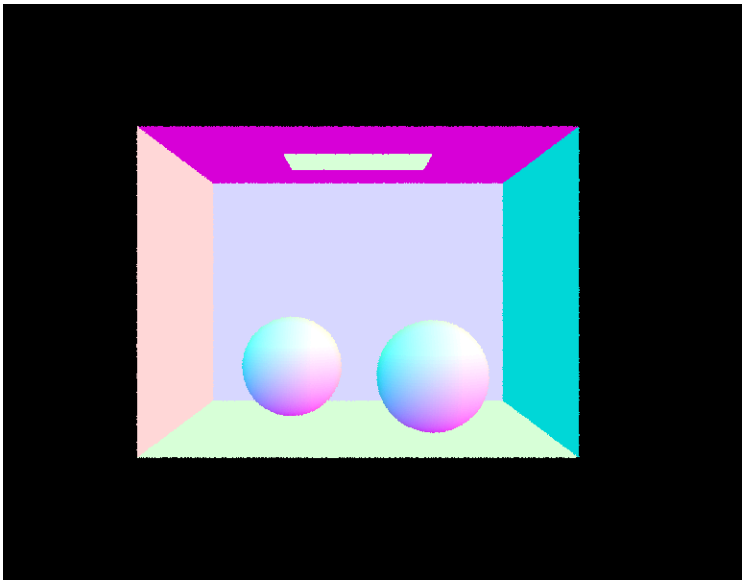
To transform the image space to the camera space, I simply use the formula on the spec, mapping (0, 0) to the corresponding bottom-left corner and (1, 1) to the upper-right corner, and everything in the middle linearly. Normalize the direction vector got from the method above.

To sample a pixel at (x, y), take a sample from the bivariate uniform distribution within (0, 0) to (1, 1) to get a uniform sample within (x, y) and (x + 1, y + 1). Calculate the pixel

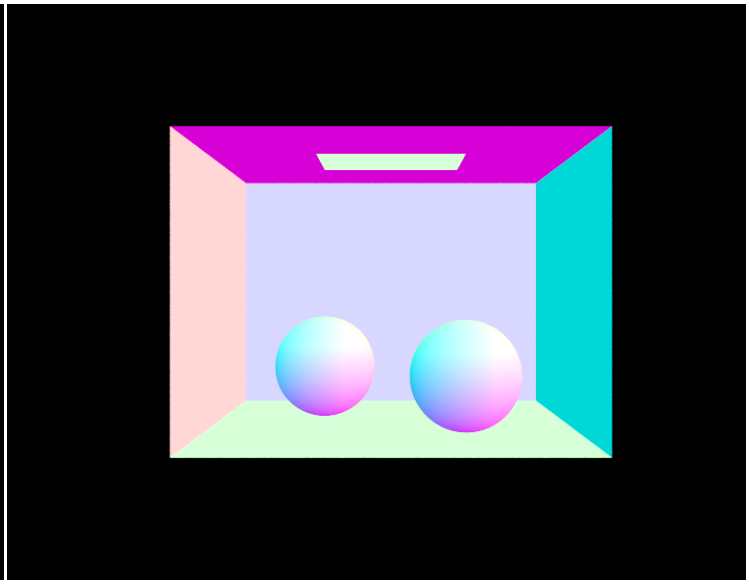
value at that location. Repeat this for a number of samples times, return the average of these samples.

To calculate the intersection of a ray and a triangle, I used the Moller Trumbore algorithm listed on the slides, basically listing all the intermediate values and calculate the needed t and normal vector. I follow the derivation formula in the slides for spheres as well.

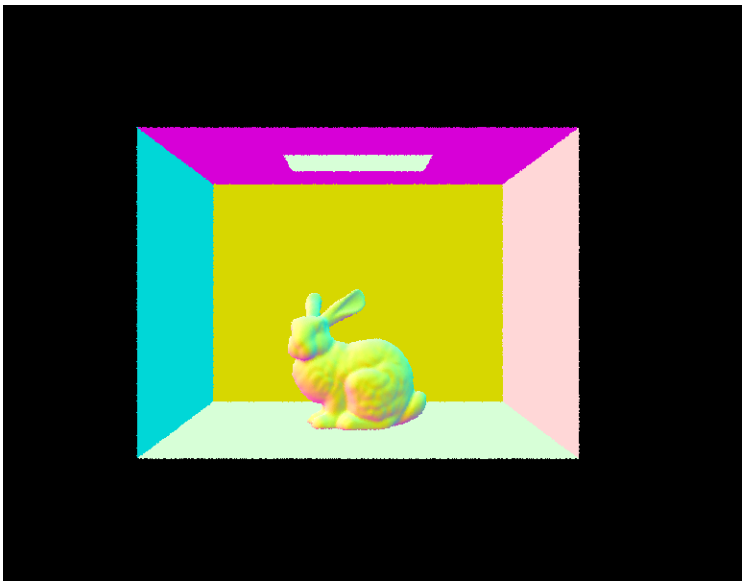
Below are the rendered images of CBspheres_lambertian.dae and CBbunny.dae with different number of sampling.



CBspheres_lambertian.dae with num_sample = 1



CBspheres_lambertian.dae with num_sample = 64



CBbunny.dae with num_sample = 1

CBbunny.dae with num_sample = 16

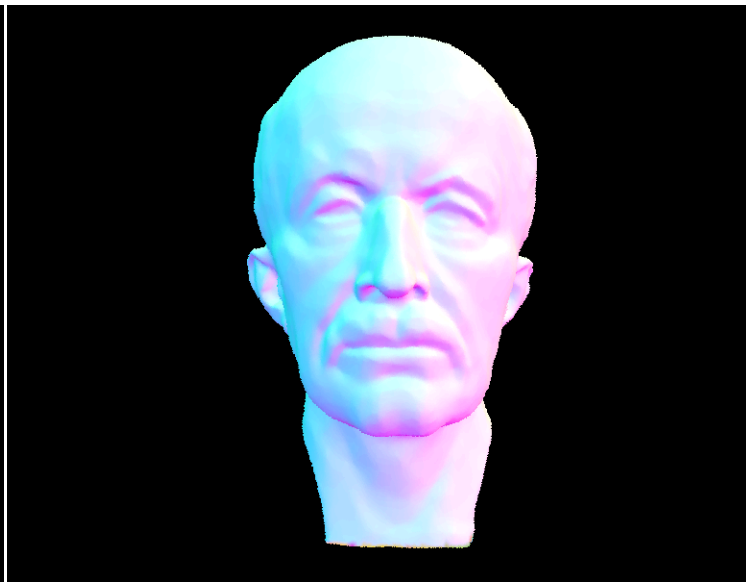
Part 2: Bounding Volume Hierarchy

To construct a BVH tree, I counted the total number of primitives in the node; if that number is greater than the

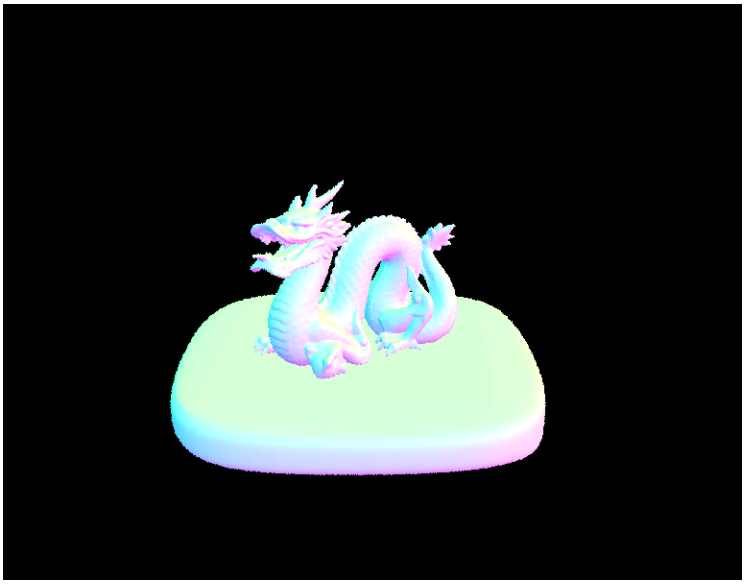
max leaf size, make three attempts to separate the primitives by the x, y, z value of the centroid and choose the one which makes the left and right node more balanced. When rendering, traverse the BVH tree from the root, check if the ray intersects with the bounding box, and only traverse through its subtree/primitives when it hits the bounding box.

Below are some large meshes that can be rendered within seconds using BVH tree acceleration.

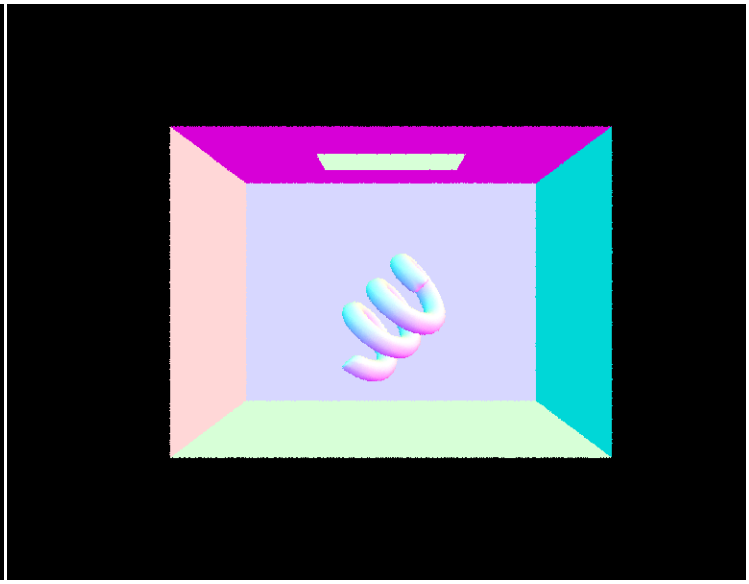
Cow



Maxplanck



Dragon



Coil

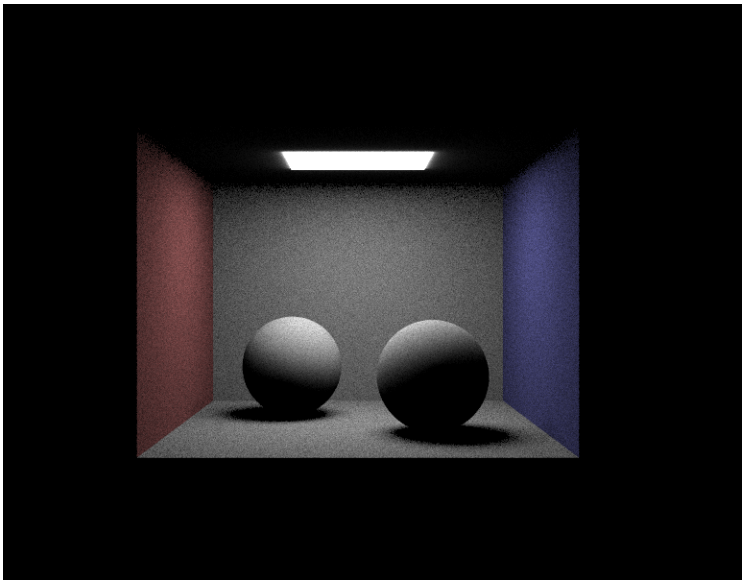
To compare the time performance with and without BVH tree acceleration, rendering a cow mesh without BVH tree would take 188 seconds, whereas it only takes 0.5 seconds with BVH tree. Similarly, rendering the coil mesh would take 108 seconds without BVH tree, compared to 4.2 seconds with BVH tree. The improvement in time performance is significant, as the BVH acceleration makes time complexity of the ray detection with primitives from O(n) to O(logn).

Part 3: Direct Illumination

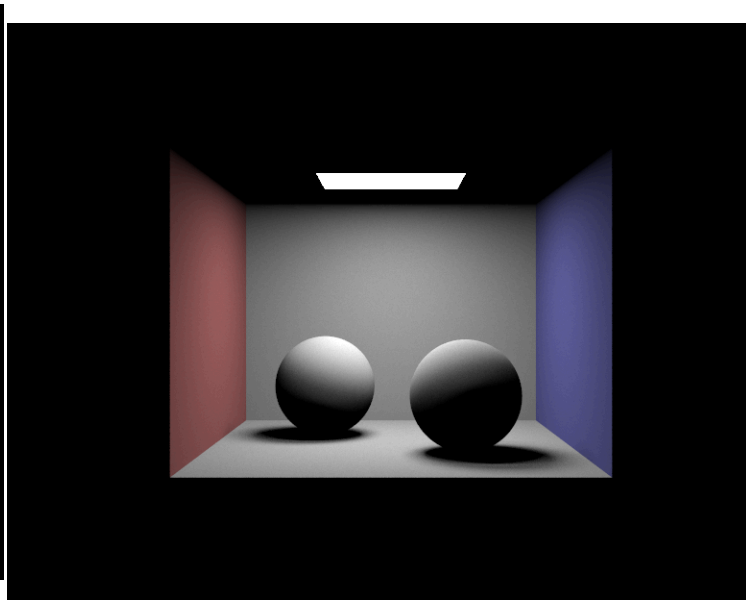
To implement direct lighting with uniform hemisphere sampling, at a given pixel and its intersection, one should sample the angles within the hemisphere around the pixel uniformly and trace the ray at that direction reversely to the other side intersection. Then get how much light there is at the intersection, and use the formula in slides to calculate how much light should be reflected to the given pixel. Average all the sampled lights got through this process.

To implement importance sampling, instead of sampling around the hemisphere, one should sample all the lights in the scene. If the sampled light ray intersects something before the given pixel, it is blocked and should be a shadow ray, thus nothing should be added to the pixel; otherwise, calculate the light casted on the pixel using the formula on the slides, and then average the samples.

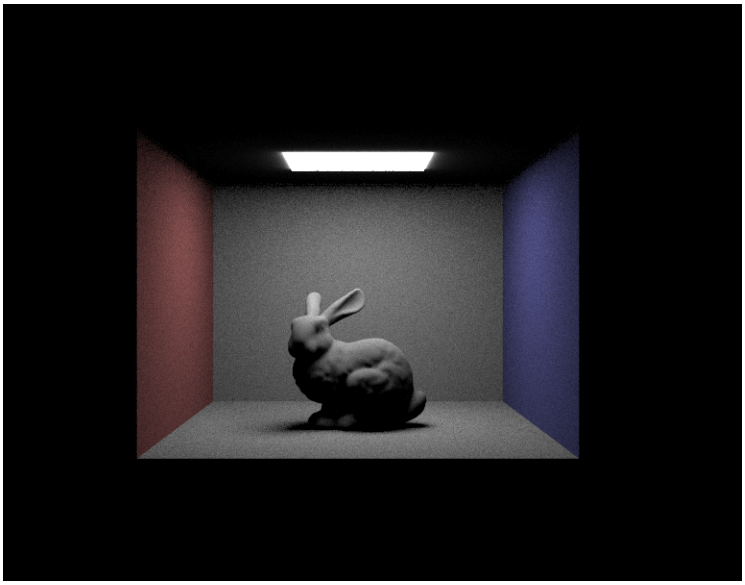
Below are the rendered images of CBspheres_lambertian.dae and CBbunny.dae, using either uniform hemisphere sampling or light sampling.



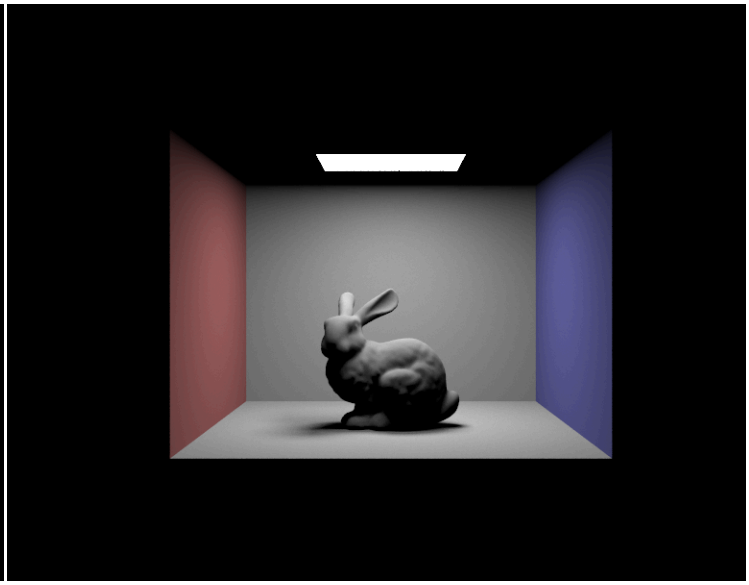
Bspheres_lambertian.dae with uniform hemisphere sampling



CBspheres_lambertian.dae with light sampling

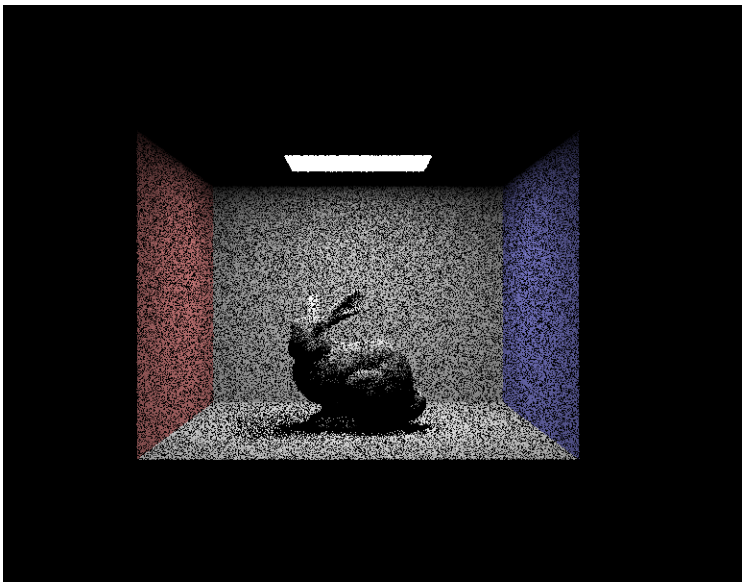


CBbunny.dae with uniform hemisphere sampling

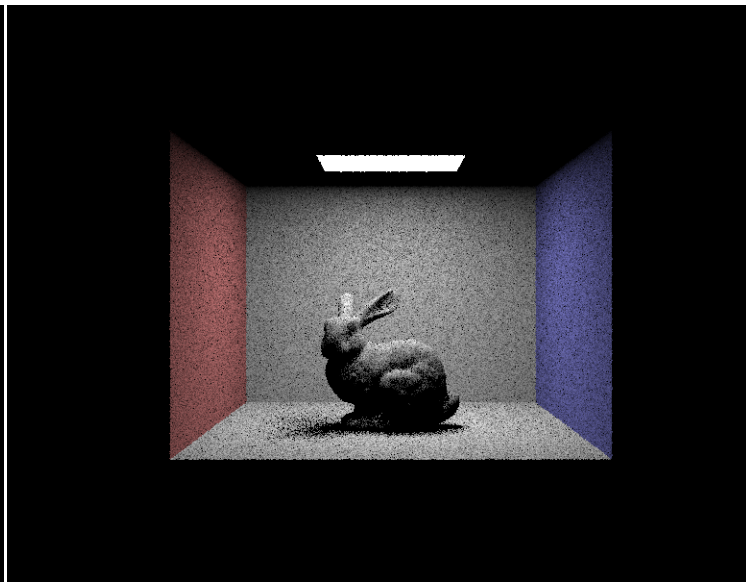


CBbunny.dae with light sampling

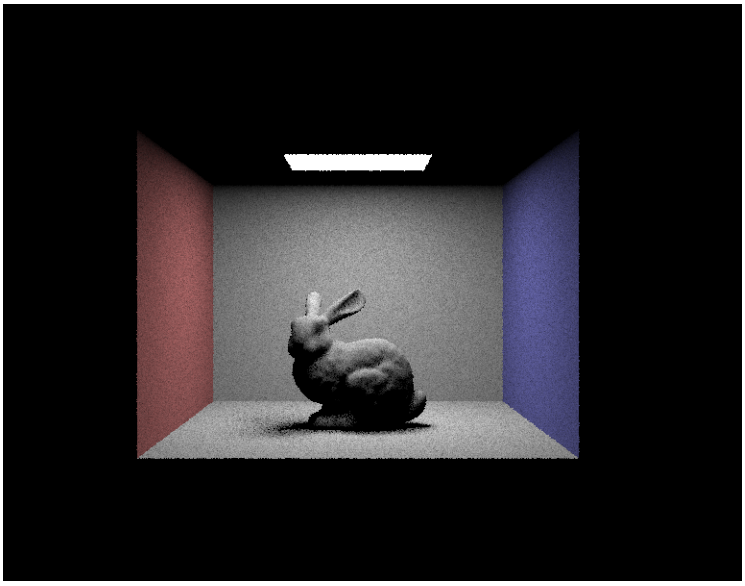
Here are the rendered images of CBbunny.dae using light sampling, but with 1, 4, 16, and 64 light rays. The improved rendering effect of less noises with of more light rays is significant in the examples, especially in the soft shadows blocked by the bunny. This effect is obvious comparing between the soft shadows with 4 light rays and that with 64 light rays, as one can tell that there are much more noises in the one with 4 light rays.



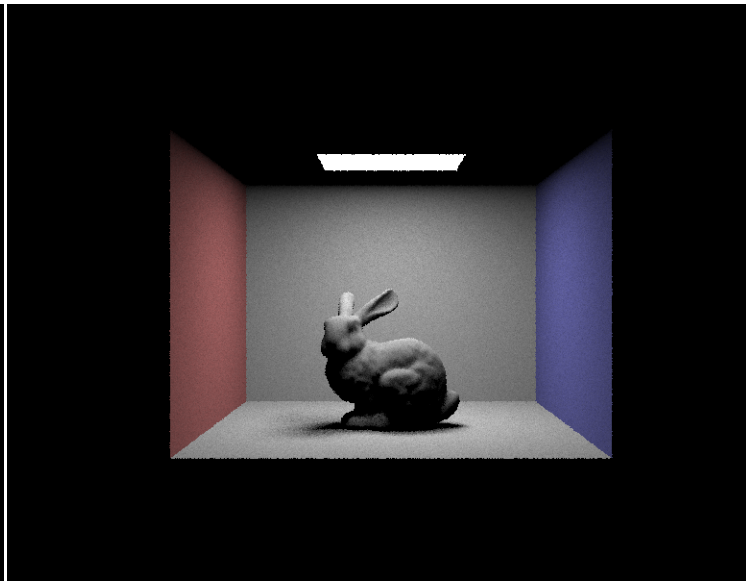
Light sampled bunny with 1 light ray



Light sampled bunny with 4 light rays



Light sampled bunny with 16 light rays



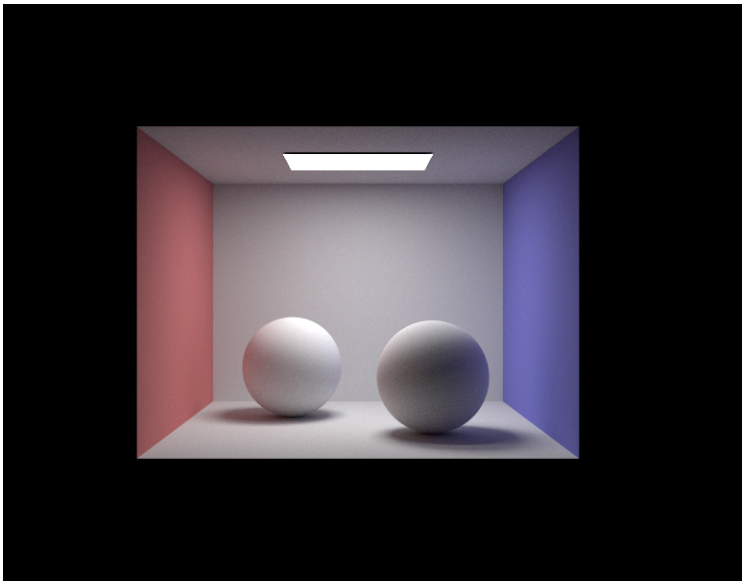
Light sampled bunny with 64 light rays

By the examples shown above, one can argue that the performance of light sampling is better than that of uniform hemisphere sampling, as light sampling produces less noise and converges faster. With a reasonable number of light rays and samples per pixel, lighting sampling produces a clean, smooth rendering, whereas uniform hemisphere sampling looks like a sheet of white noise is added to the image generated.

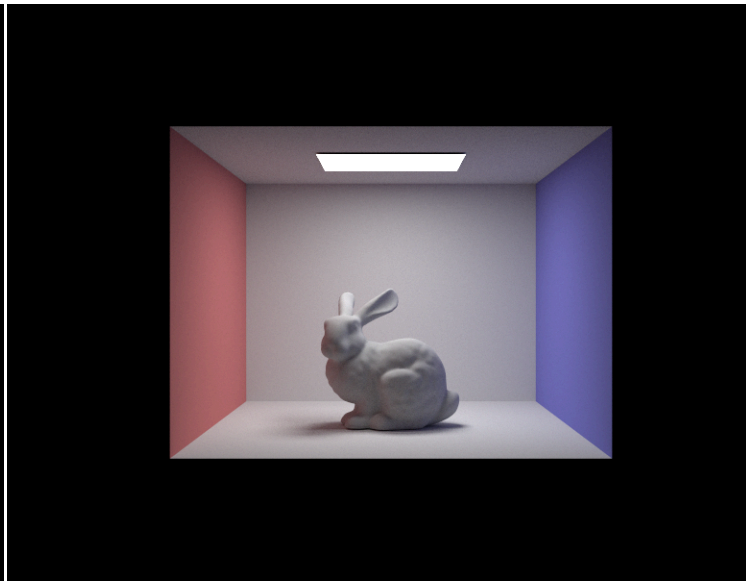
Part 4: Global Illumination

To implement indirect lighting, one should first keep record of the current depth of ray objects. Then, for lighting at a given depth, return the one-time bounce radiance at that depth, calculate the reflected new ray, and recursively call the lighting of that ray with a depth + 1, add together the results of all calls until the max ray depth is reached. If the russian roulette technique in which early termination with probability is implemented, for each layer, terminate the summation with the termination probability, and scale up the result with 1 - termination probability when not terminated; end the recursive summation when the max ray depth is reached before early termination as well.

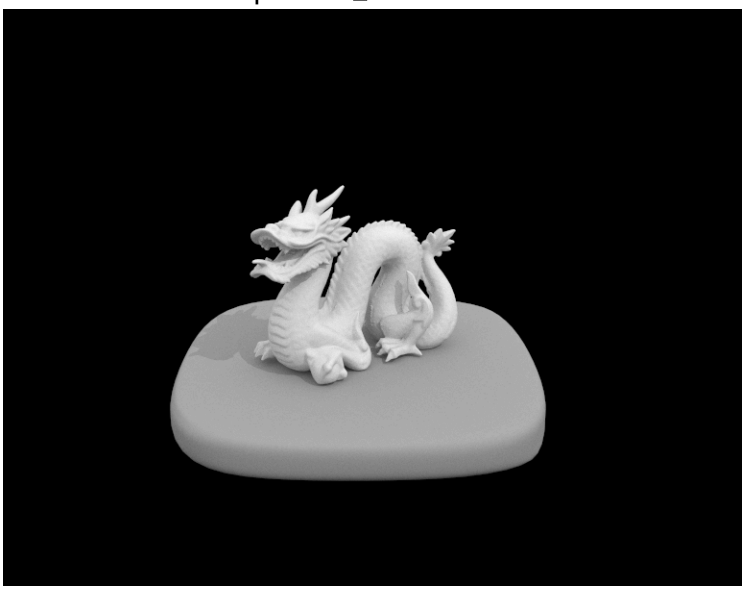
Below are some images rendered with direct and indirect illumination, all with 1024 samples per pixel, and max ray depth of 5.



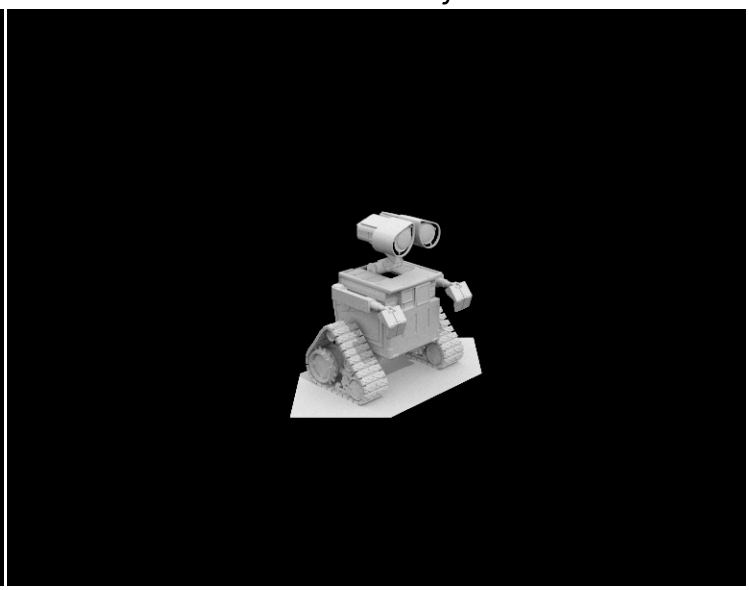
CBspheres_lambertian



CBbunny

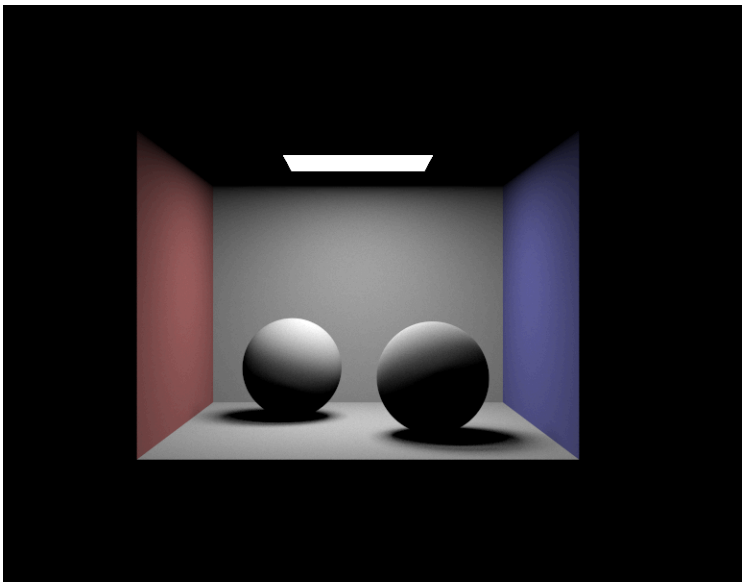


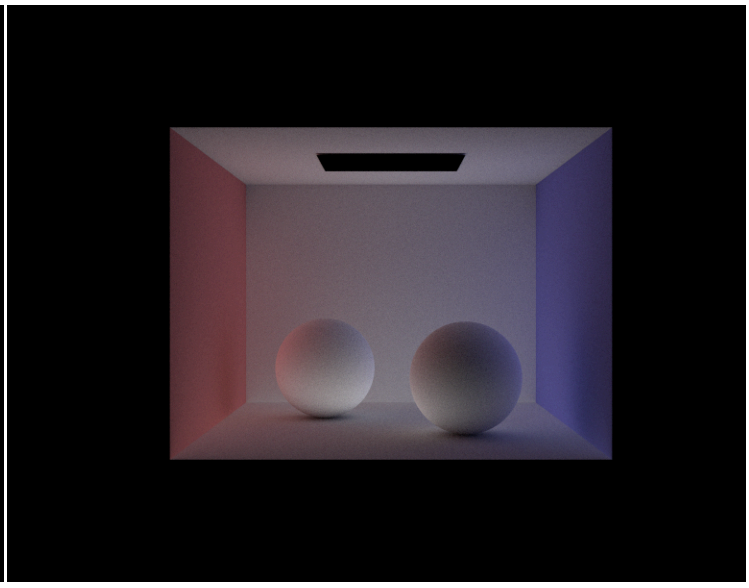
Dragon



Wall-e

Below are images of CBspheres_lambertian.dae with only direct lighting and indirect lighting.





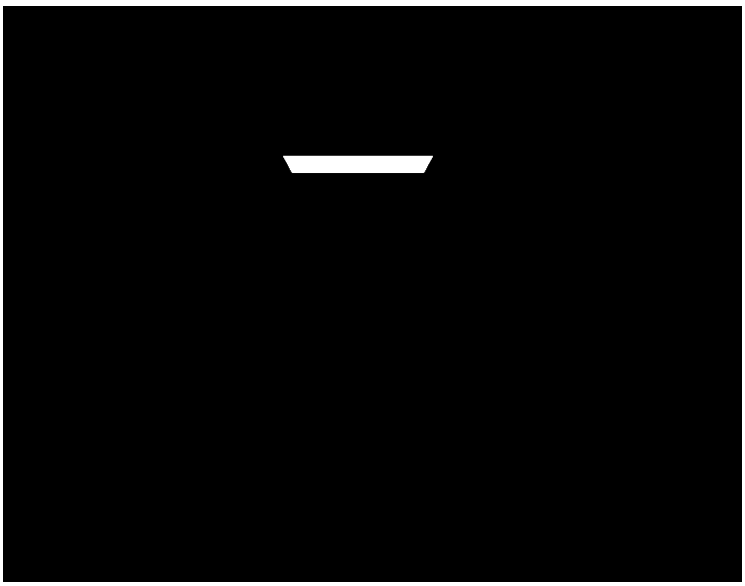
CBspheres_lambertian with direct lighting CBspheres_lambertian with indirect lighting

Here are the renders of CBbunny with max layer depth m = 0 to m = 5. On the left, there are rendering with only rays at that depth (o = 0); on the right all rays of the current depth and previous depths are summed up during rendering (o = 1).

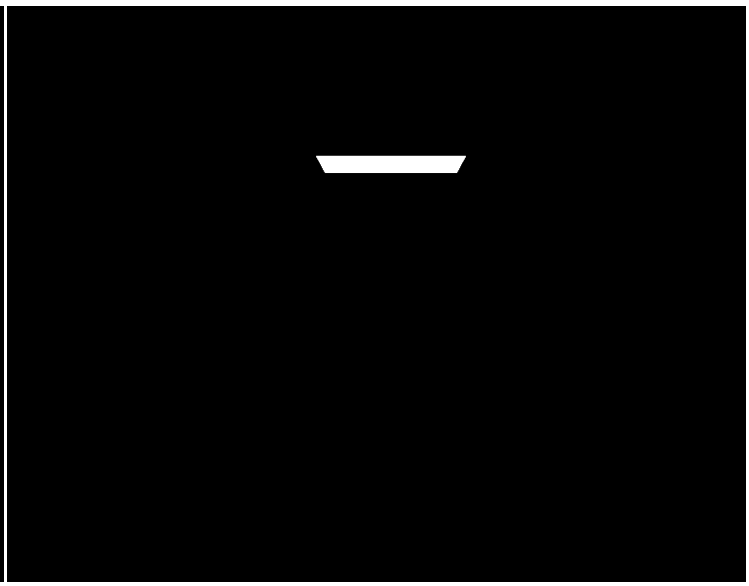
Taking a close look on the 2nd bounce of light only, it seems to lighten the area where the the rays would not reach directly, mainly the ceiling and the shadow area under the bunny. Adding the 2nd bounce causes the image to produce the soft shadow effect to these regions.

For the 3rd bounce of light, it seems to be a general illumination of the entire scene, as most regions can be reached by three bounces of rays. Adding it to the image slightly make the entire image more illuminated, enriching the illumination of the whole image.

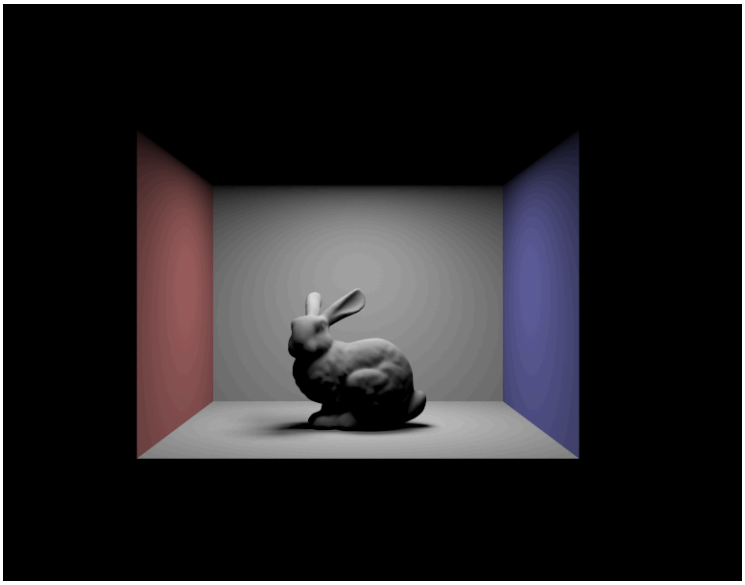
As the max ray depth m increases, the effect of rays on that depth decreases, and the entire rendering tends to converge.



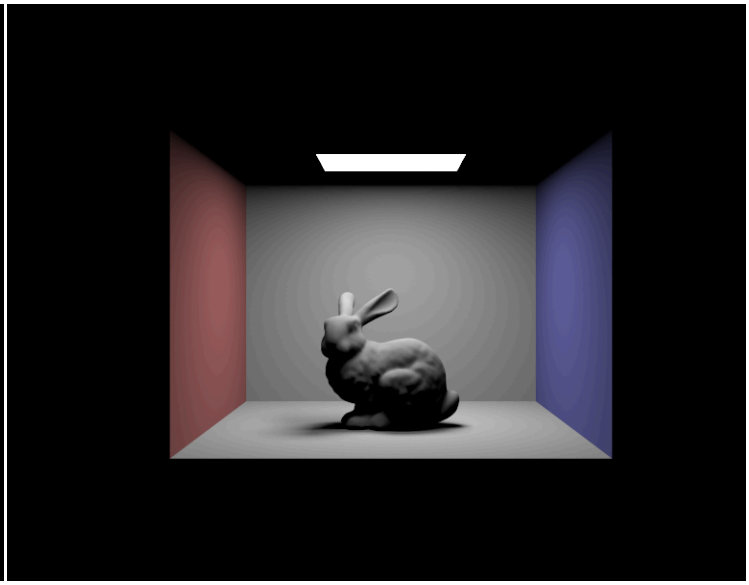
m = 0, o = 0



m = 0, o = 1

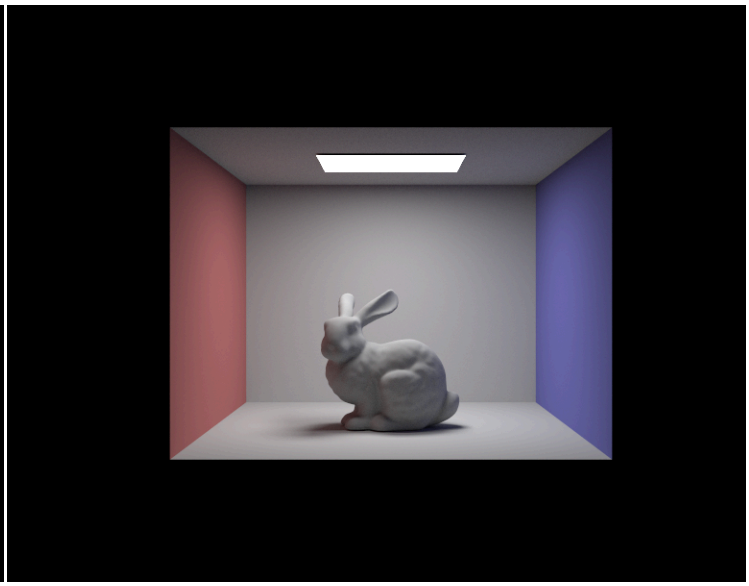


m = 1, o = 0

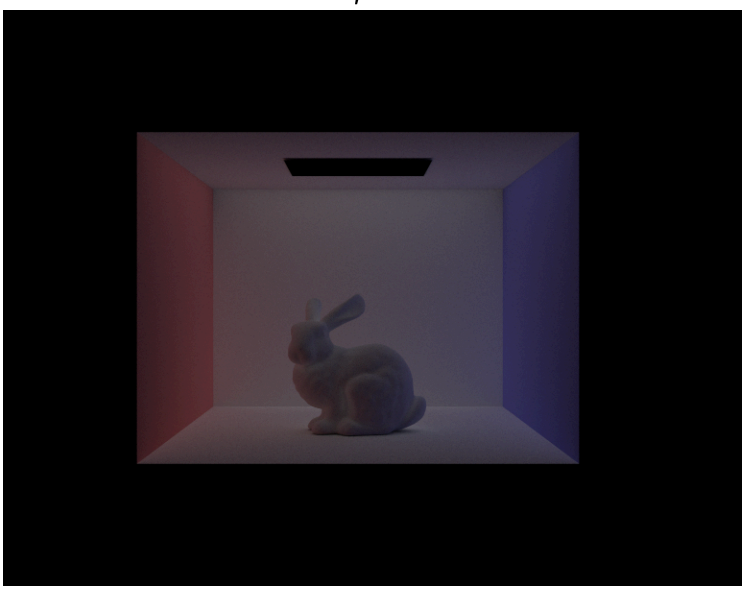


m = 1, o = 1

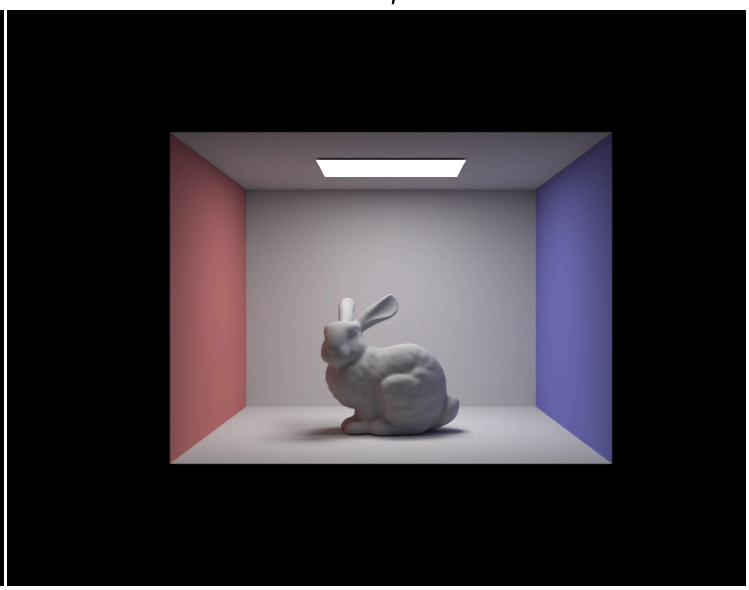
m = 2, o = 0



m = 2, o = 1

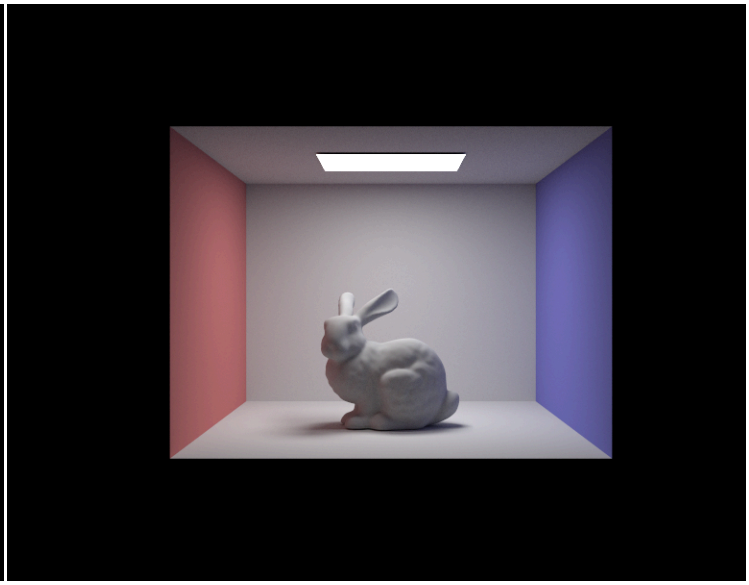


m = 3, o = 0

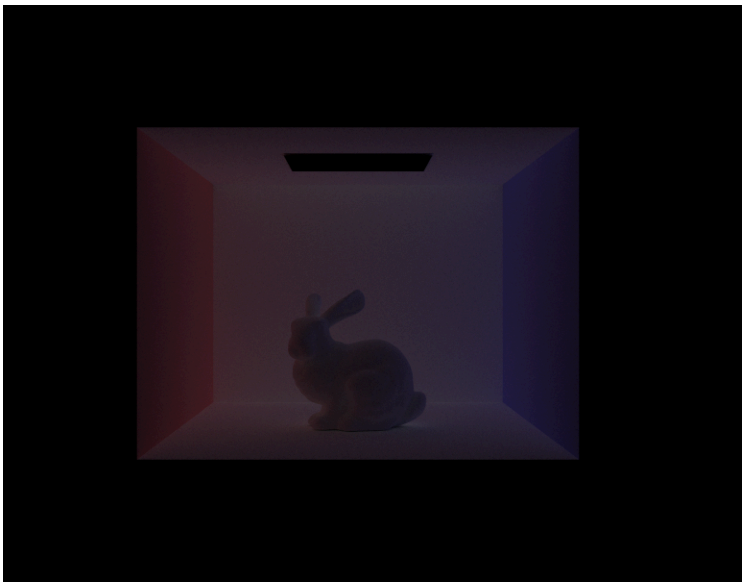


m = 3, o = 1

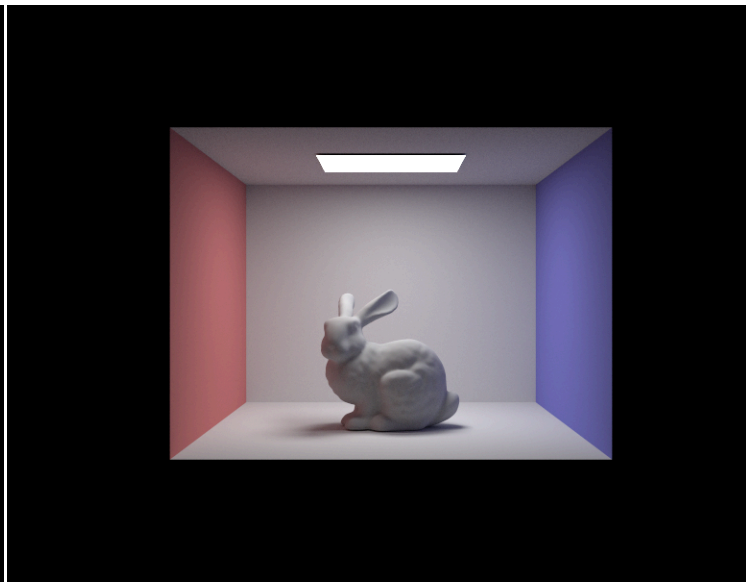
m = 4, o = 0



m = 4, o = 1

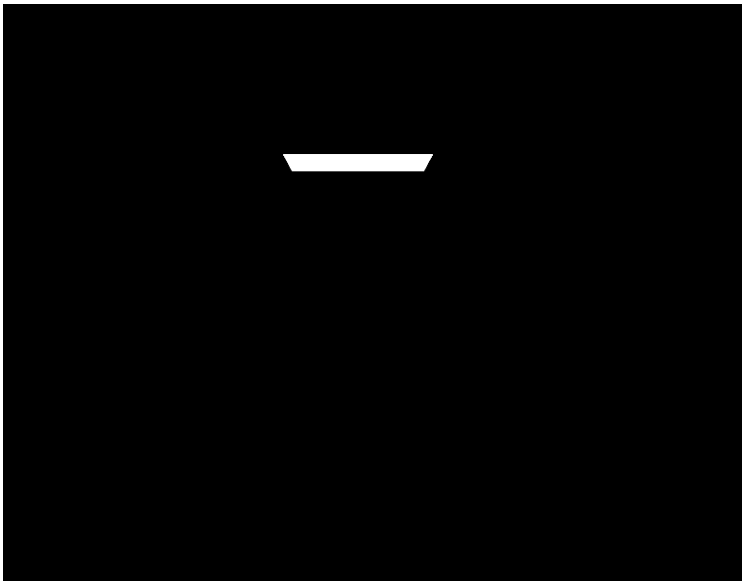


m = 5, o = 0

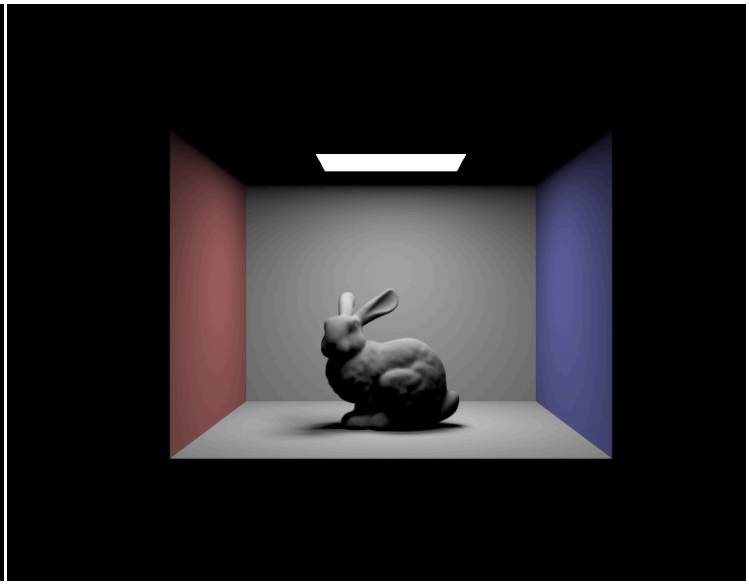


m = 5, o = 1

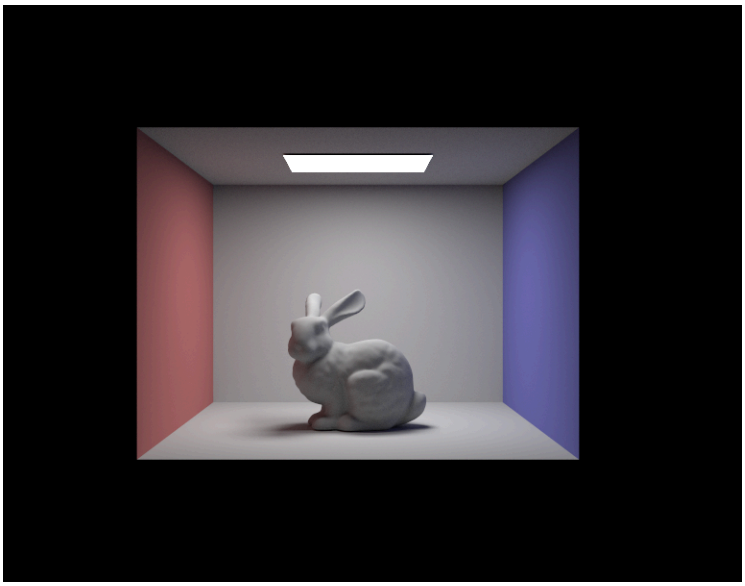
To see the effect of Russian Roulette with different max ray depth, here are the rendering of CBbunny with m = 0, 1, 2, 3, 4, and 100 with 1024 pixels per sample. Tracing rays with may ray depth of 100 layers add a general illumination to the whole image compared to 4 layers.



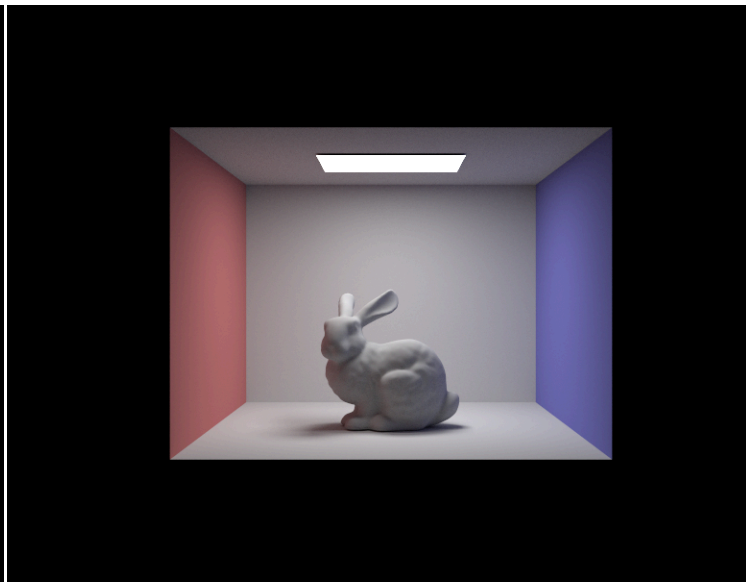
m = 0



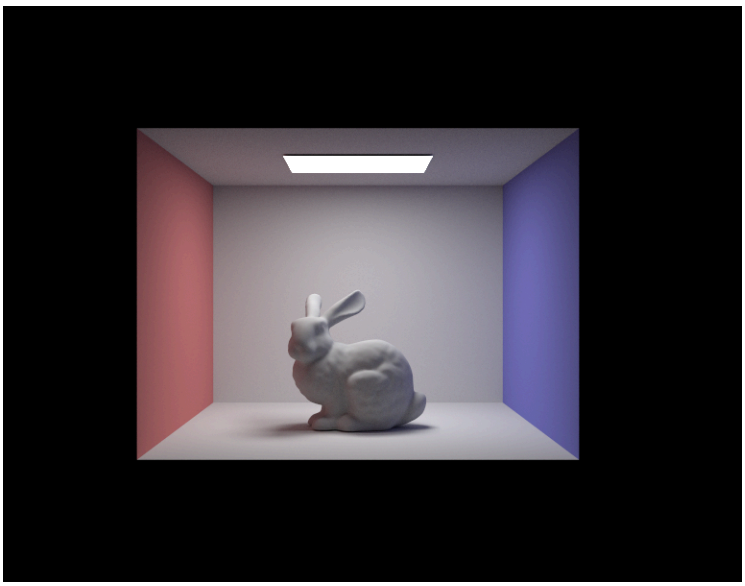
m = 1



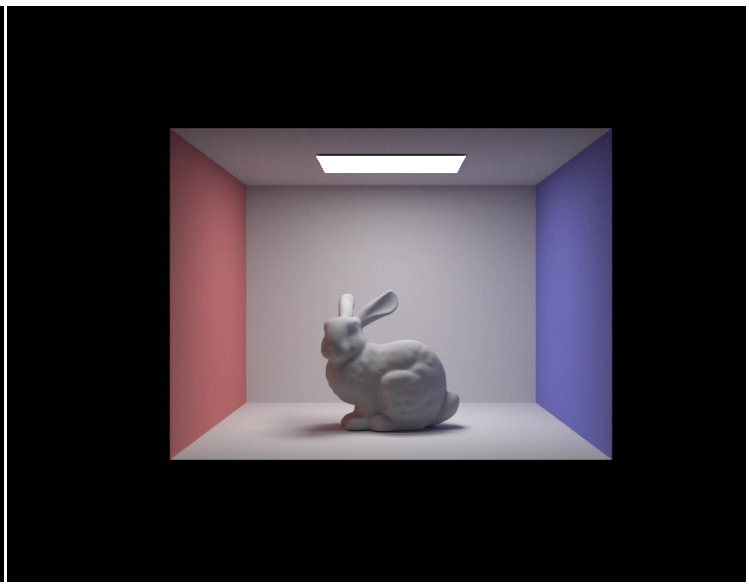
m = 2



m = 3



m = 4

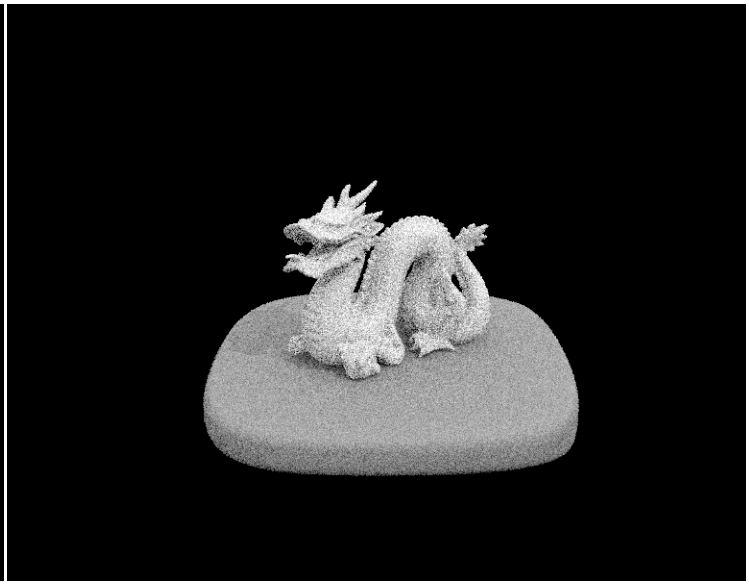


m = 100

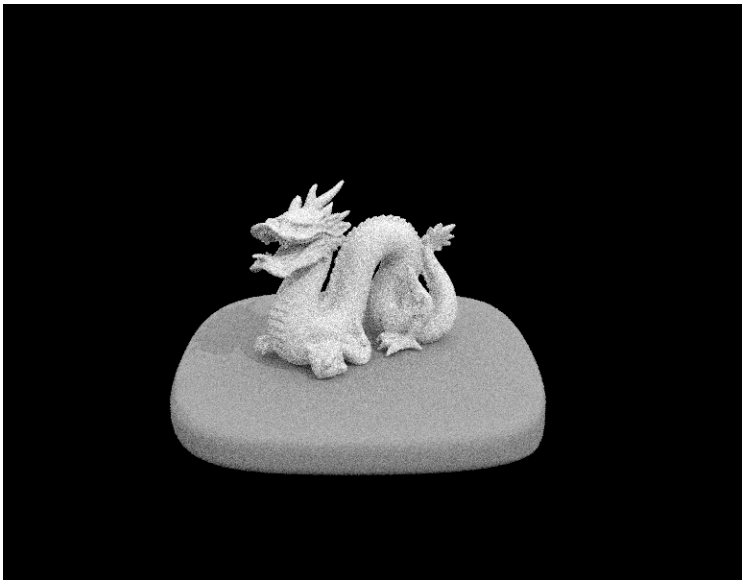
Here are the rendered views of the dragon mesh with different sample per pixel rates, and 4 light rays. Based on

these images, the effect of greater number of sample per pixel rates decreases the noise and improve the overall quality of the images.

Dragon with 1 sample per pixel

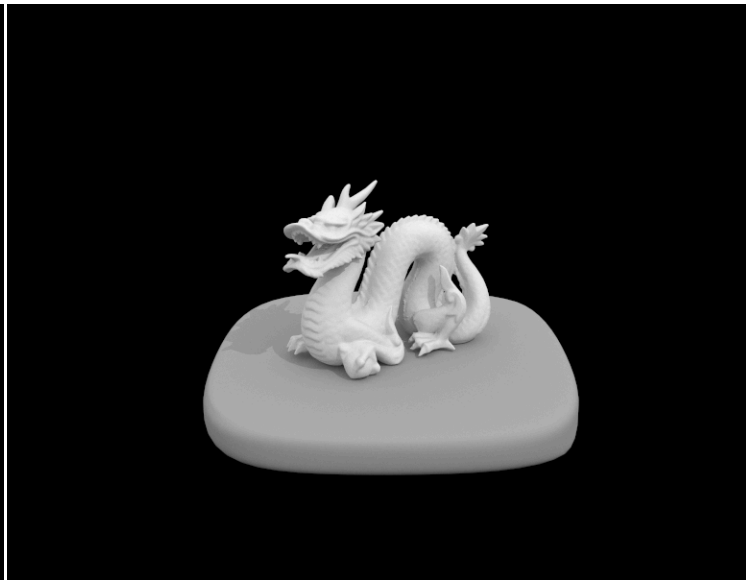


Dragon with 2 sample per pixel



Dragon with 4 sample per pixel

Dragon with 16 sample per pixel



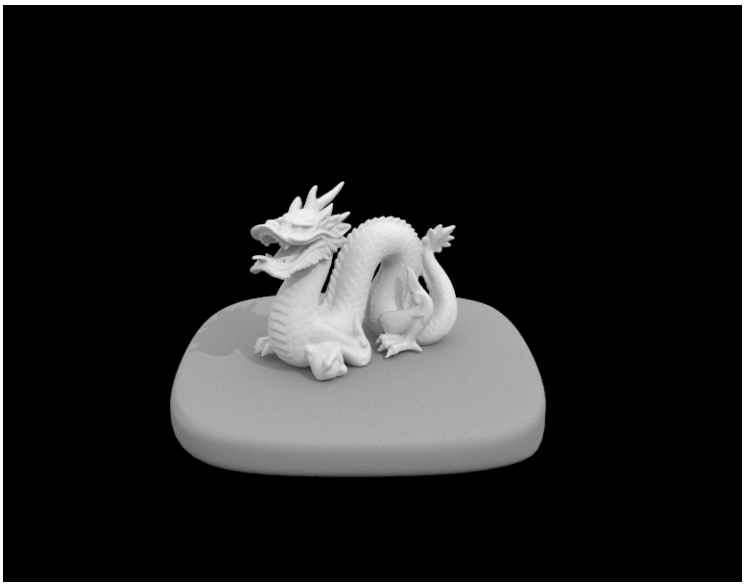
Dragon with 64 sample per pixel

Dragon with 1024 sample per pixel

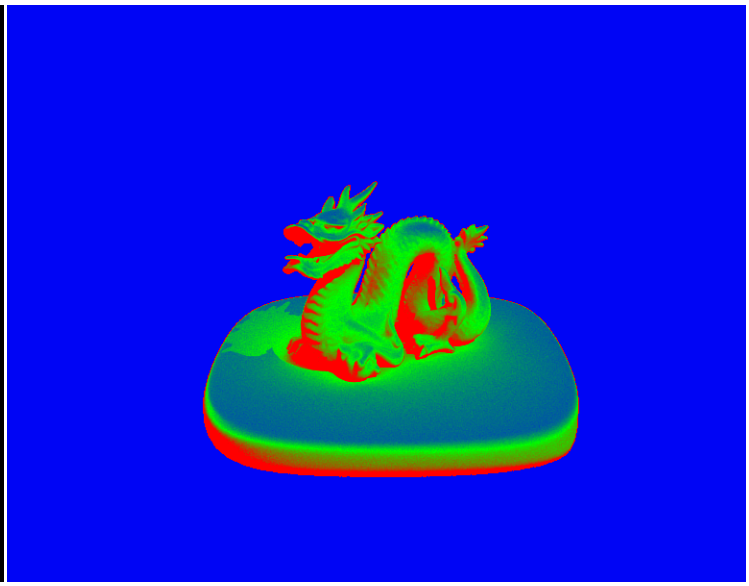
Part 5: Adaptive Sampling

Adaptive sampling is a technique which reduces the time spent on rendering by applying less samples on pixels that converge faster and more samples on those converge slower. To calculate the convergence of a pixel, defined in the spec, record the sum of illuminance and sum of squared illuminance of the current samples. Then, after a given number of samples per batch, calculate the convergence of the pixel with n samples within O(1) time to determine whether this pixel has pretty much converged already. If yes, stop early.

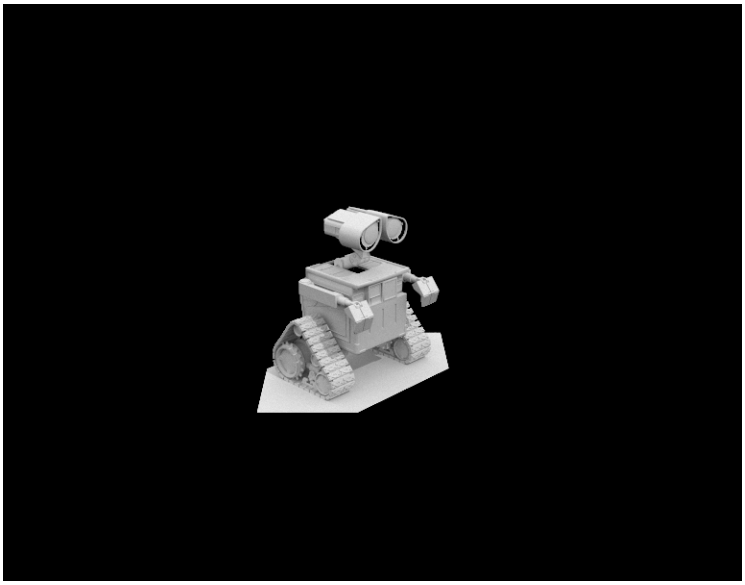
Below are the rendered images and the sampling rate images of dragon and wall=e with adaptive sampling, at 2048 samples per pixel. From the sampling rate images, pixels with lower sampling variance it would take fewer samples to converge.



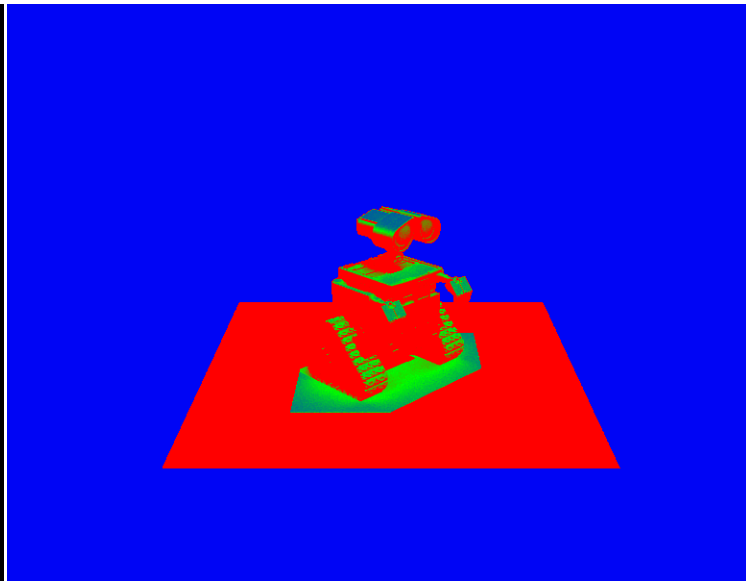
Dragon with adaptive sampling



Dragon sampling rate



Wall-e with adaptive sampling



Wall-e sampling rate