CS 184 HW Write Ups

link to webpage: https://cal-cs184-student.github.io/hw-webpages-sp24-EmmanuelCobian/hw3/index.html

Homework 3 - Pathtracer

In this assignment, I implemented the core routines of a physically-based renderer using a pathtracing algorithm. I implemented concepts like ray-scene intersection, acceleration structures, and physically based lighting and materials. I really enjoyed seeing the realistic looking pictures that I was able to generate by the end of this assignment. I also found the last part on adaptive sampling really interesting because while the concept seemed a bit abstract and complicated at first, the implementation wasn't all that tough and instead pretty intuitive.

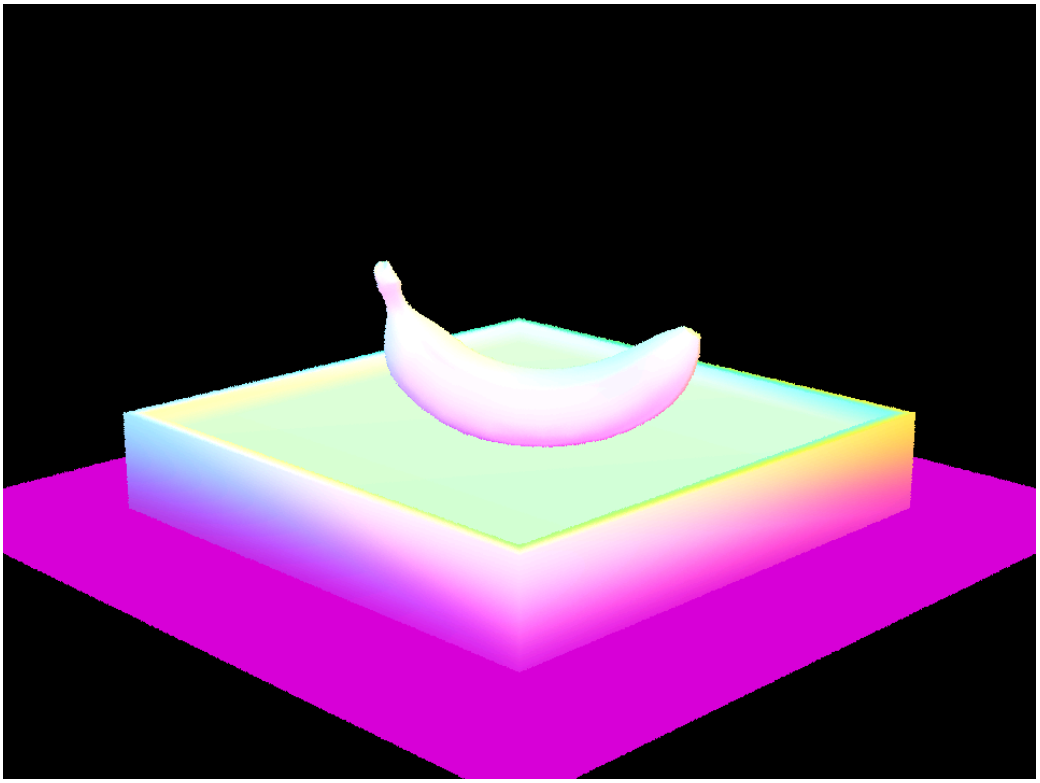
Part 1: Ray Generation and Scene Intersection

Walk through of the ray generation and primitive intersection parts of the rendering pipeline

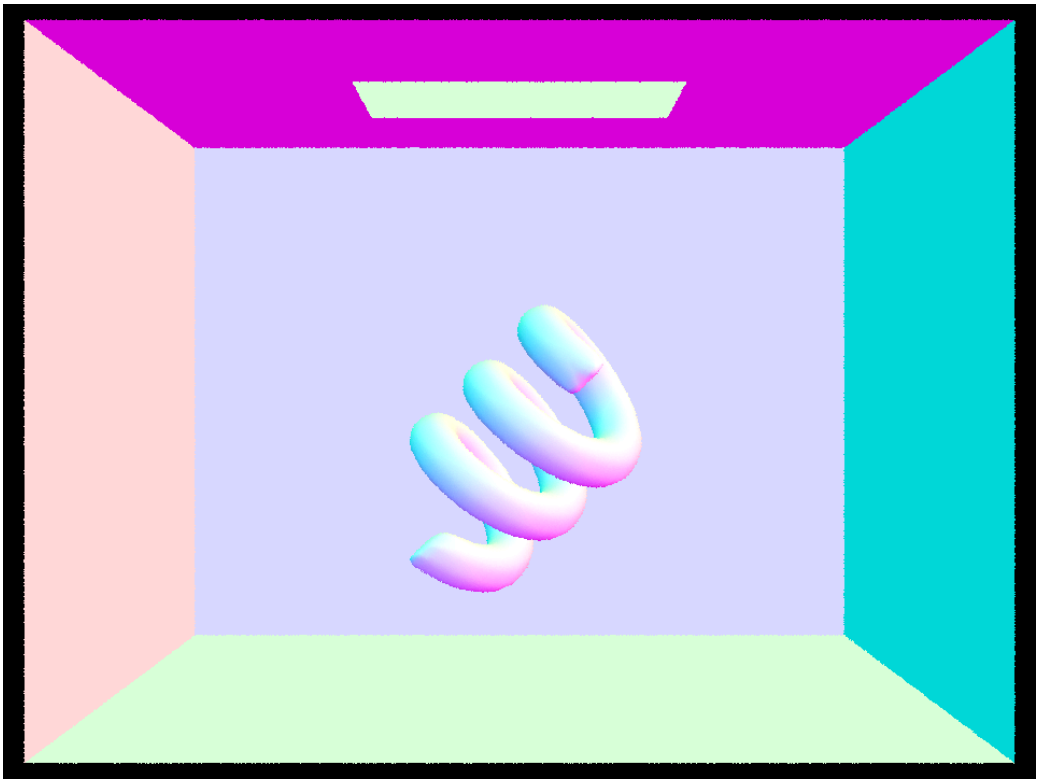
inside `raytrace_pixel()`, I wrote a loop that calls `camera->generate_ray(...)` to get camera rays and `est_radiance_global_illumination(...)` to get the radiance along those rays. When generating rays, I sample based off a unit square to get random samples from a given ray origin and after averaging them, I write that value to the sample buffer.

How was triangle intersection algorithm implemented?

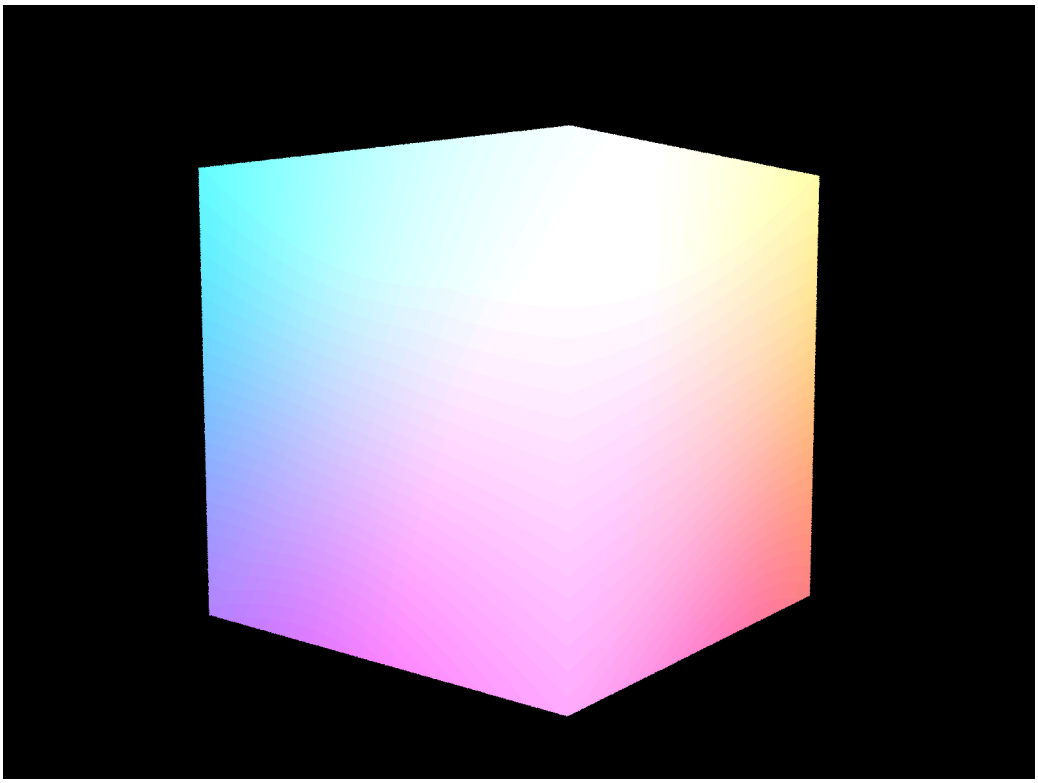
By implementing the efficient version of computing triangle intersections from lecture, I add the logic of finding the time, `t`, when the ray intersects a triangle. If that value is within the established bounding parameters `max_t` and `min_t`, then I can say that a particular intersection is valid. After finding out whether an intersection occurs, I update `max_t` and fill in the fields for the `Intersection` object like the time of intersection, normal vector, primitive, and the bsdf.



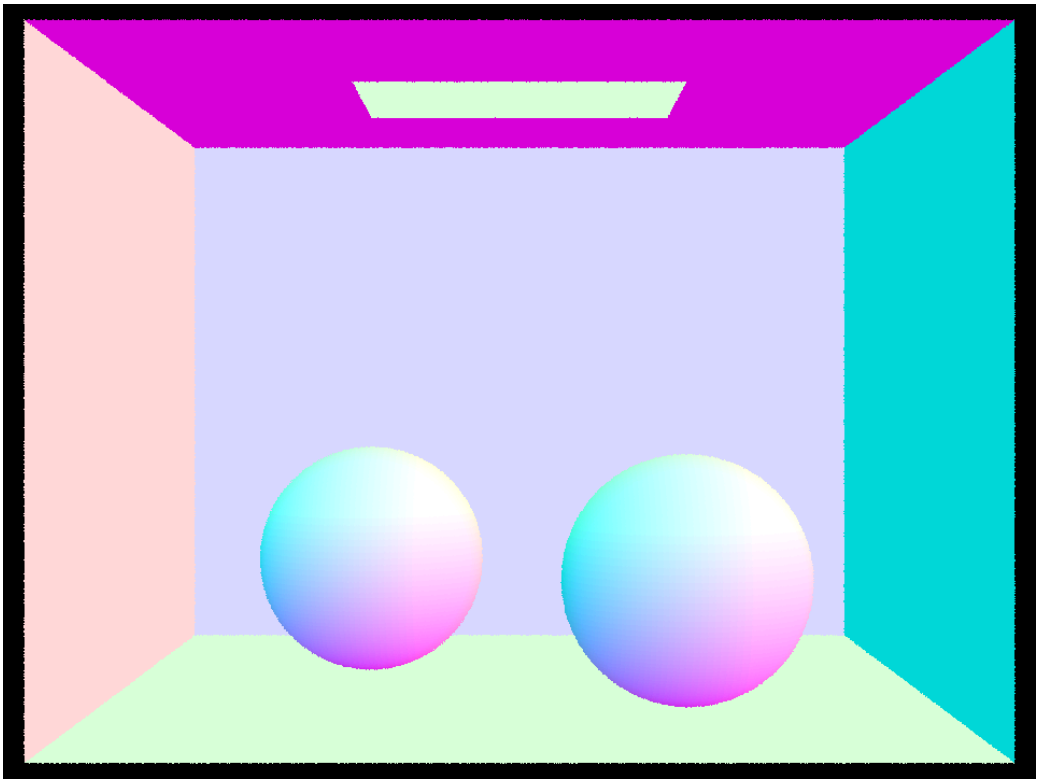
Bannana figure with normal shading



Coil figure with normal shading



Cube figure with normal shading



Sphere figures with normal shading

Part 2: Bounding Volume Hierarchy

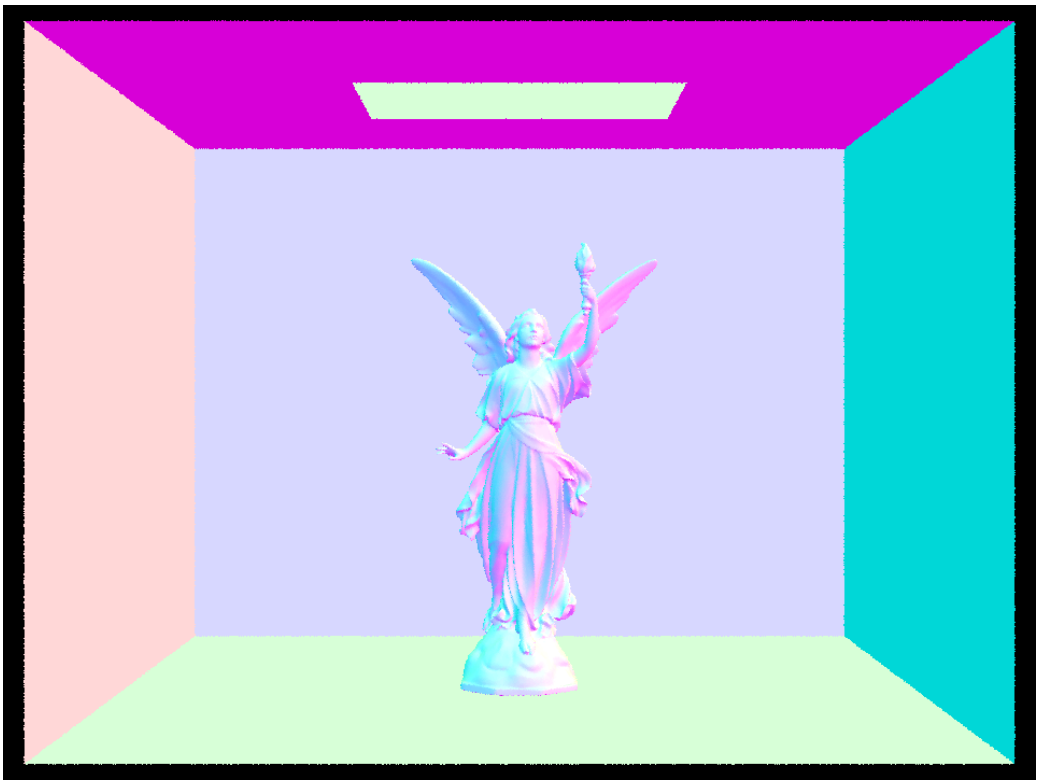
Walk through BVH construction algorithm.

I start out by creating a new BBox and adding all the current primitives in it. After, I put this `BBox` inside a `BVHNode` instance and check whether this node has less than `max_leaf_size` number of primitives. If it does, then this is considered a leaf node and I return it with the appropriate assignments of `start` and `end`. If the node contains more primitives, then I split them into left and right partitions and recursively assign the left and right children of the parent `BVHNode`.

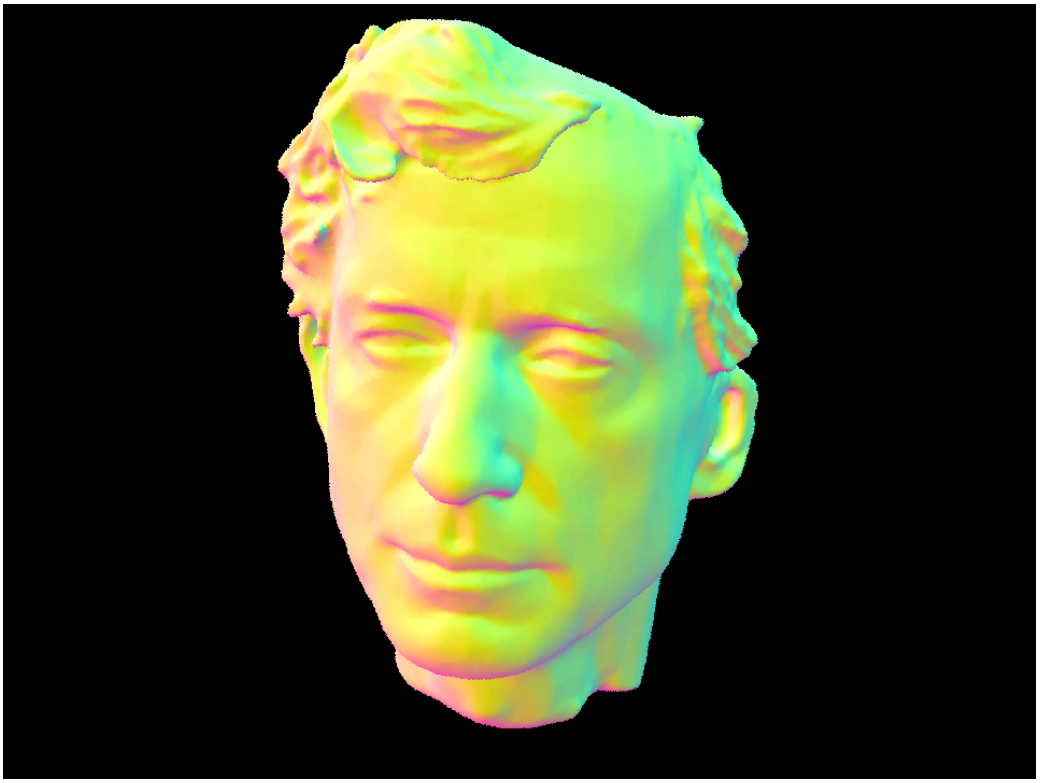
What heuristic was used for picking the splitting point?

I decided to split the primitives along the middle axis that has the smallest value of `<l, l> + <r, r>` where `l` and `r` are the `BBox` elements containing left and right primitives. These primitives were split on the mean of each axis' centroid values. I found this heuristic

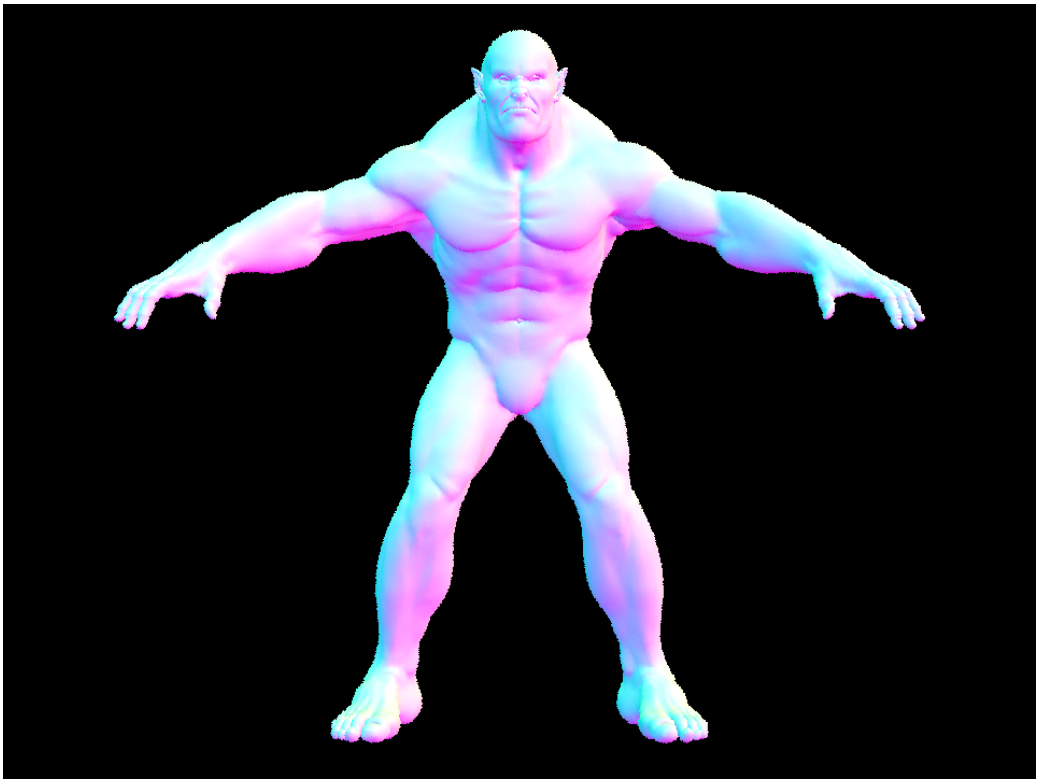
sort of by trial and error. I initially had a simple y-axis split heuristic but it caused a lot of slow down when rendering scenes in later parts of this hw, so I came back to this and tried out a lot of combinations until I landed on this one that seems to perform pretty well.



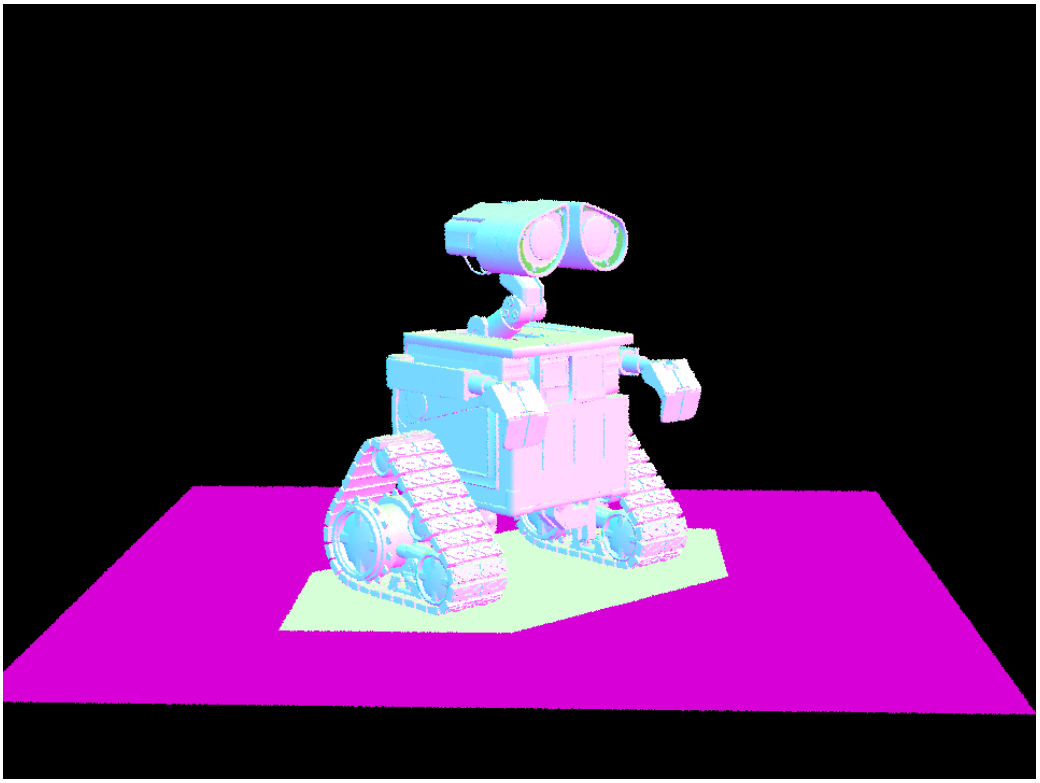
CBlucy.dae with normal shading & BVH



peter.dae with normal shading & BVH



beast.dae with normal shading & BVH



wall-e.dae with normal shading & BVH

Compare rendering times on a few scenes with moderately complex geometris with and without BVH acceleration.

When rendering the scenes with BVH acceleration, rendering times were magnitudes better. For example, when rendering the CBlucy.dae model, the rendering time was **189.8450s** without BVH acceleration and **0.1983s** with BVH acceleration. Another example, with the peter.dae scene, the rendering time was **16.9840s** without BVH acceleration and **0.5076s** with BVH acceleration. This is a significant improvement and it's clear that BVH acceleration is a good technique to use when rendering complex scenes.

By definition, it makes sense that the speed up is so large when using BVH acceleration because for a scene, the program only needs to check for intersections in areas where it crosses bounding boxes. This optimization allows us to short circuit a lot of computation since it's unlikely that a ray will intersect a majority of the primitives.

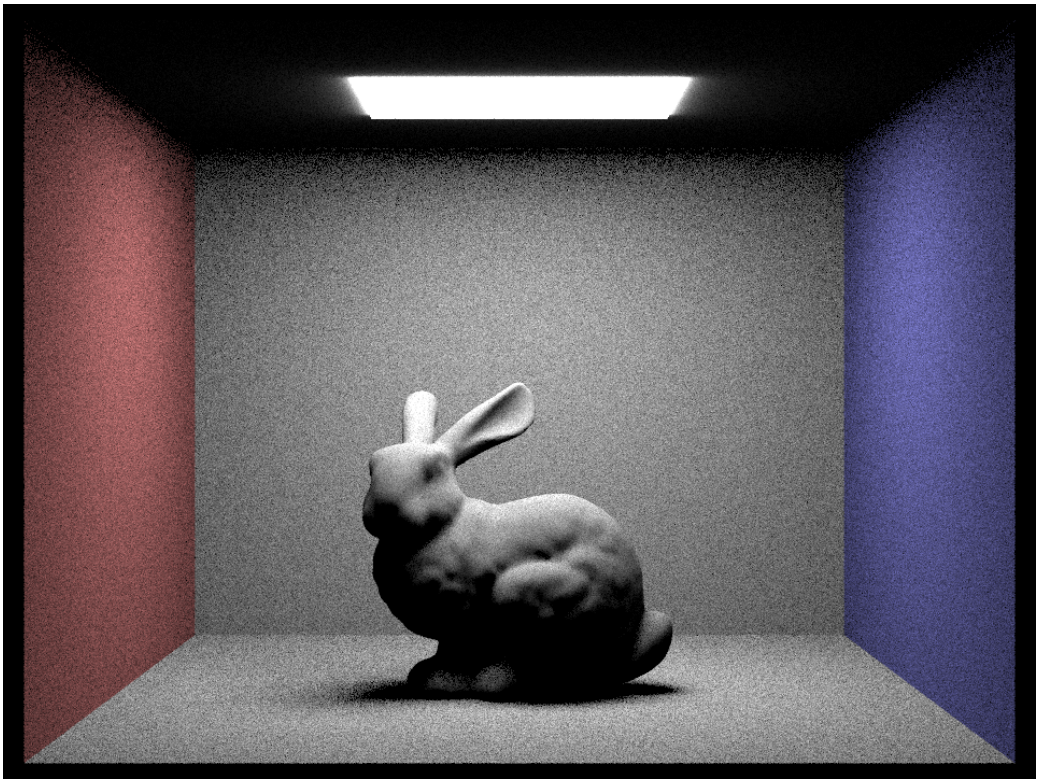
Part 3: Direct Illumination

Walk through both implementations of the direct lighting function

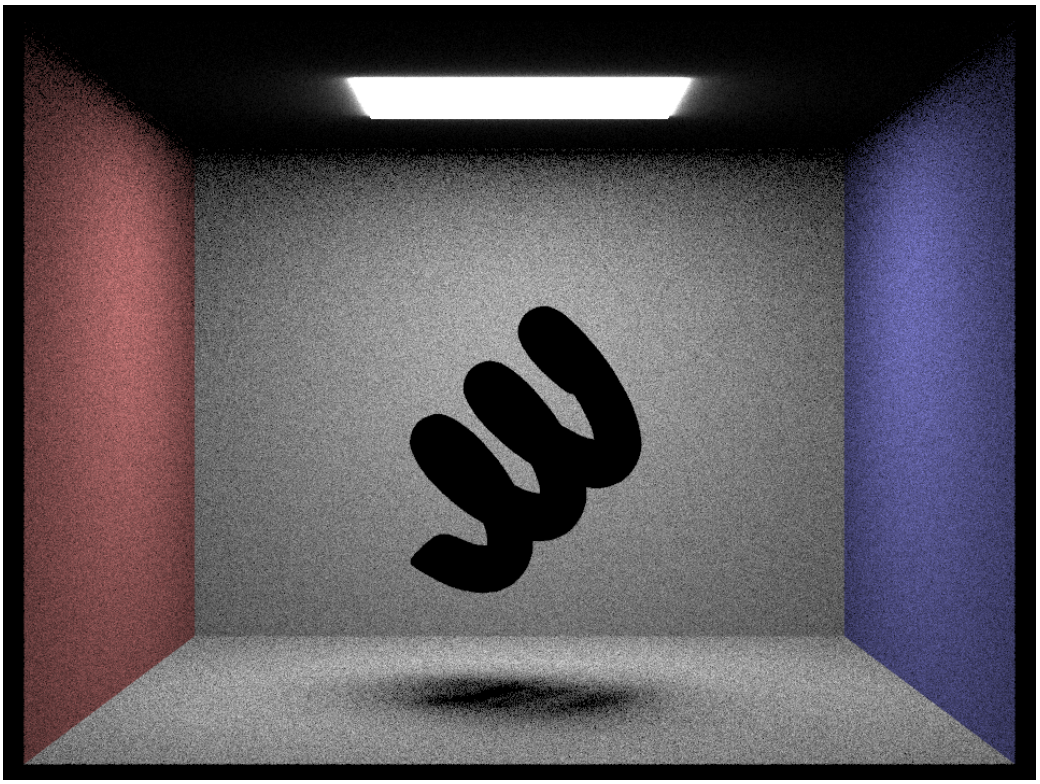
For hemisphere sampling, I first establish the probability density function as `1/PI`. Afterwards, I draw uniform `num_samples` from the hemisphere using this pdf. With each sample, `wi`, I find the bsdf of the intersection between the sample and `w_out`. Then, I build a new ray with `hit_p` as the origin and `wi` as the direction. If it's the case that this casted ray intersects the `bvh`, then this sample is added to the total contribution `L_out`. Finally, this term is normalized by the number of samples and the pdf. On the other hand, lighting sampling isn't all that different from hemisphere sampling. Now instead of just sampling uniform `num_samples`, we draw samples from the lights directly. So now, I add an outer for loop that iterates over the light sources in the scene. The implementation is pretty much the same from there.

Images rendered with both implementations of the direct lighting function

Uniform Hemisphere Sampling

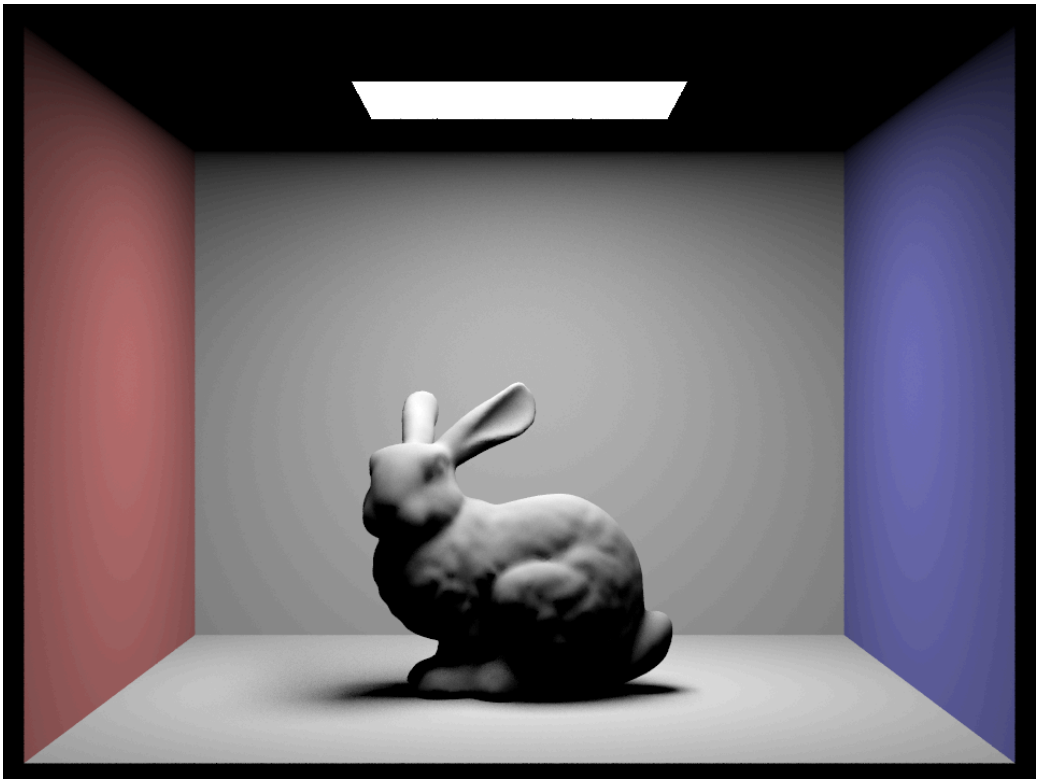


bunny.dae

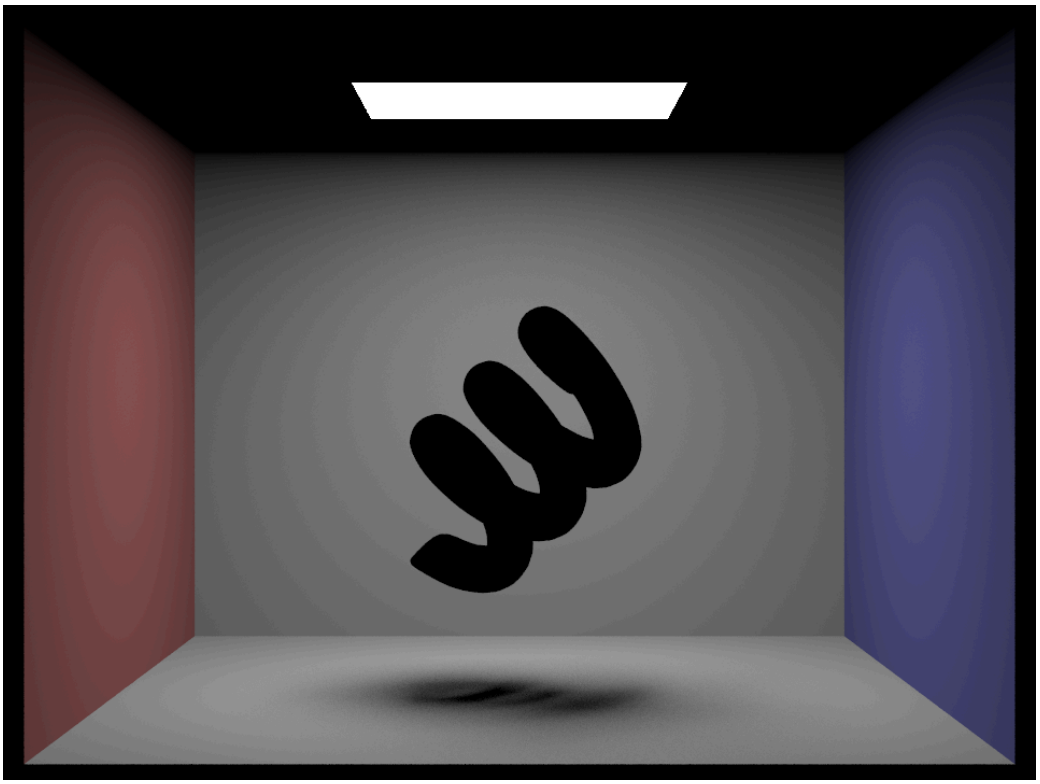


coil.dae

Light Sampling



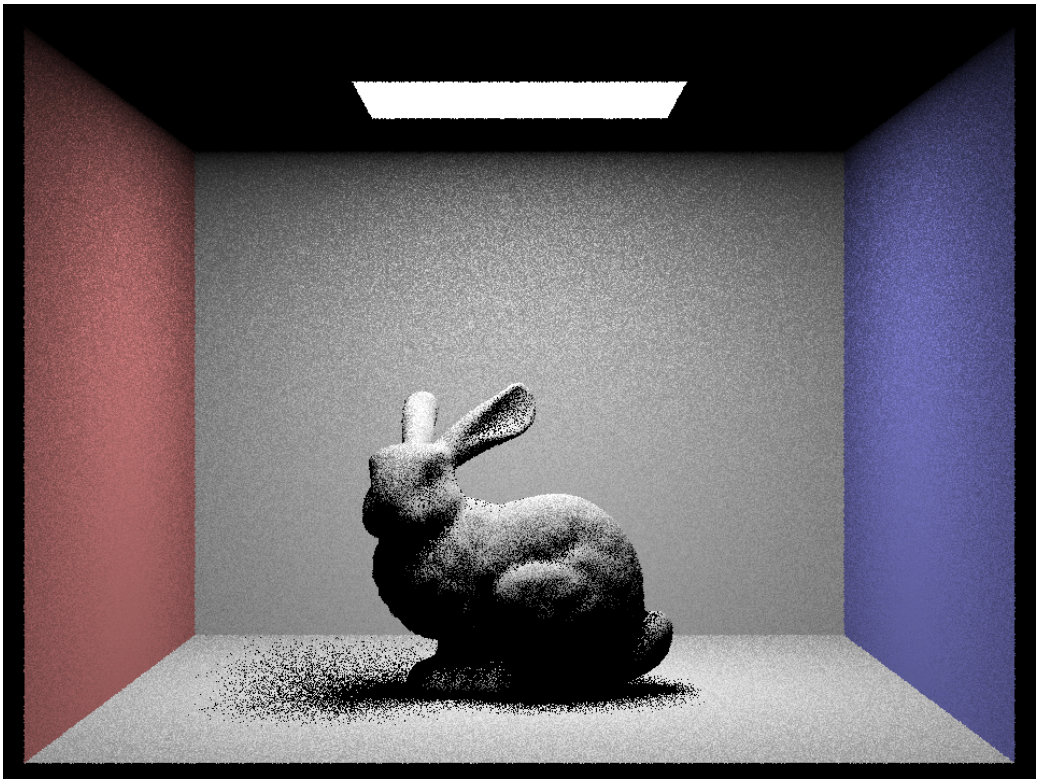
bunny.dae



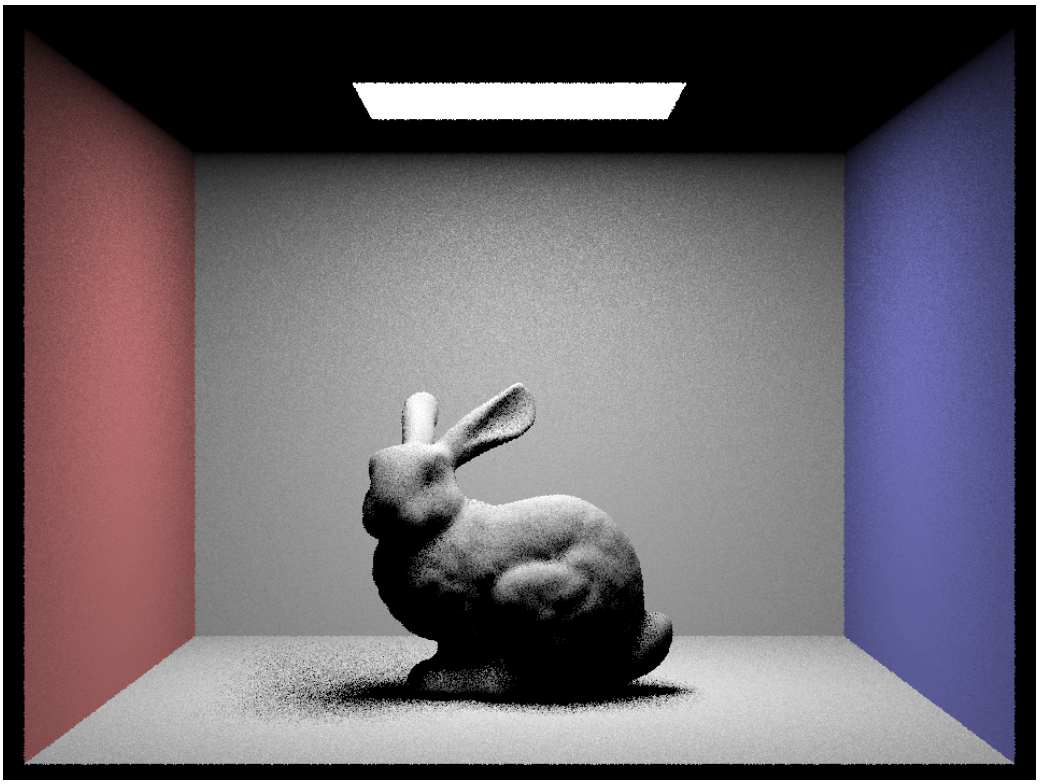
coil.dae

Soft shadow noise comparison with different light ray values

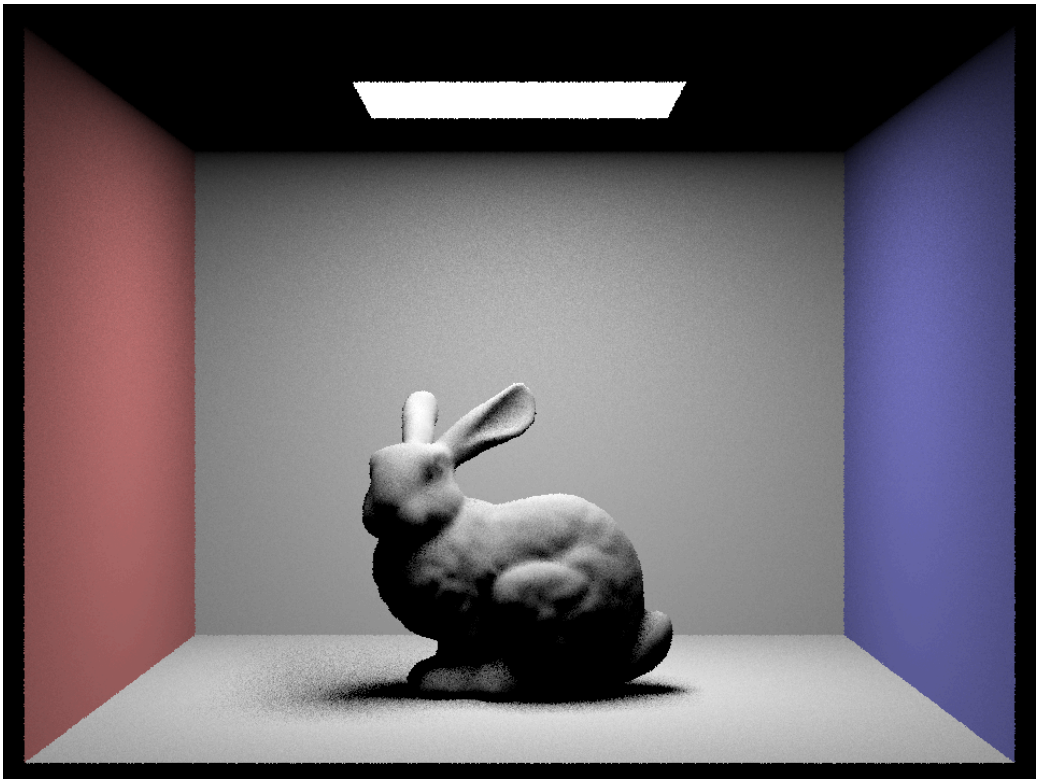
I notice a general trend that with higher light ray counts comes less soft shadow noise. When looking at 1 light ray vs 64 light rays, the difference is night and day. Soft shadows for 1 light ray are very scattered, it's easy to see the harsh noise on the ground below the bunny's haed. When there's 64 light rays, however, it's much harder to even notice any noise at first glance.



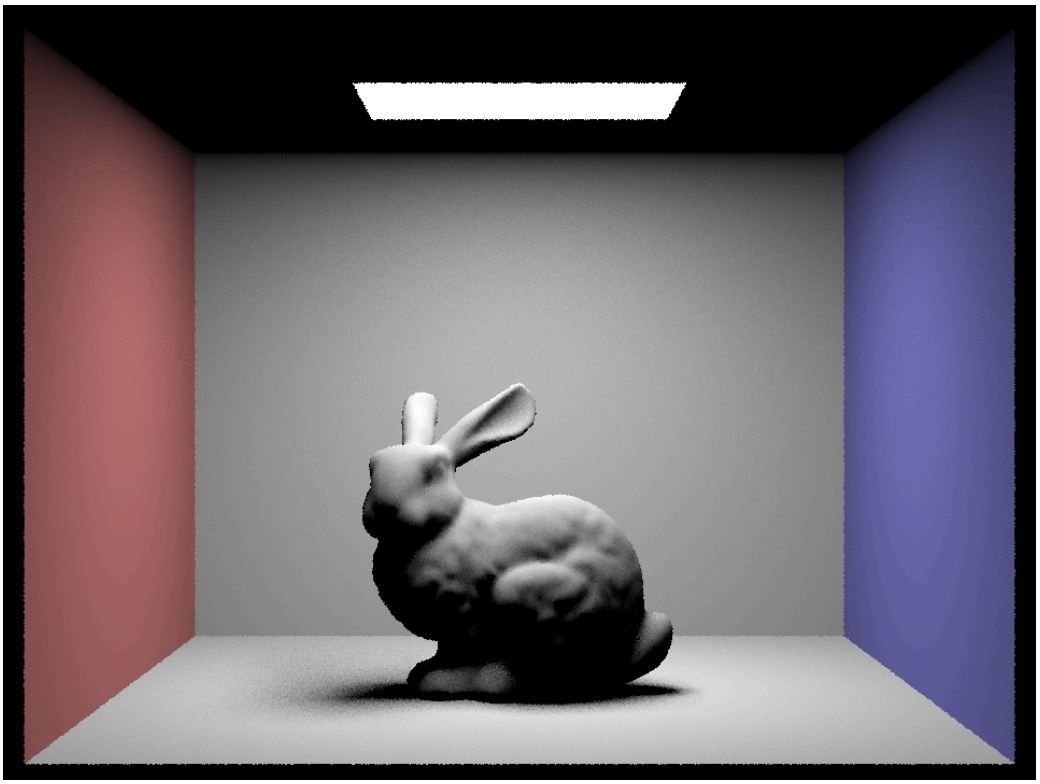
1 light ray



4 light rays



16 light rays



64 light rays

Uniform hemisphere sampling vs lighting sampling results

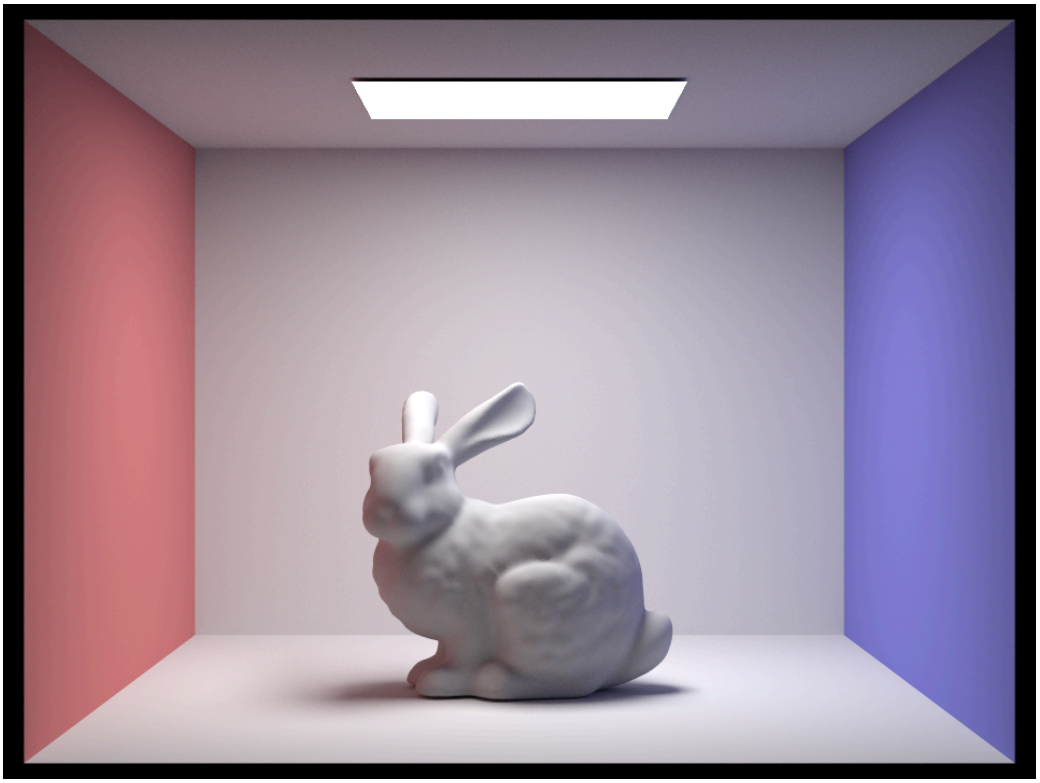
Overall, I would say that when using the same number and values of parameters for the different types of sampling, lighting sampling is always better. While hemisphere lighting does eventually converge, it takes a larger value of rays and samples per ray to get similar results as light sampling with fewer samples and rays. Intuitively, this makes sense because with light sampling, we narrow the scope of where we sample to only the directions that face a light source. On the other hand, hemisphere lighting is just pointing in an random direction within the entire scene.

Part 4: Global Illumination

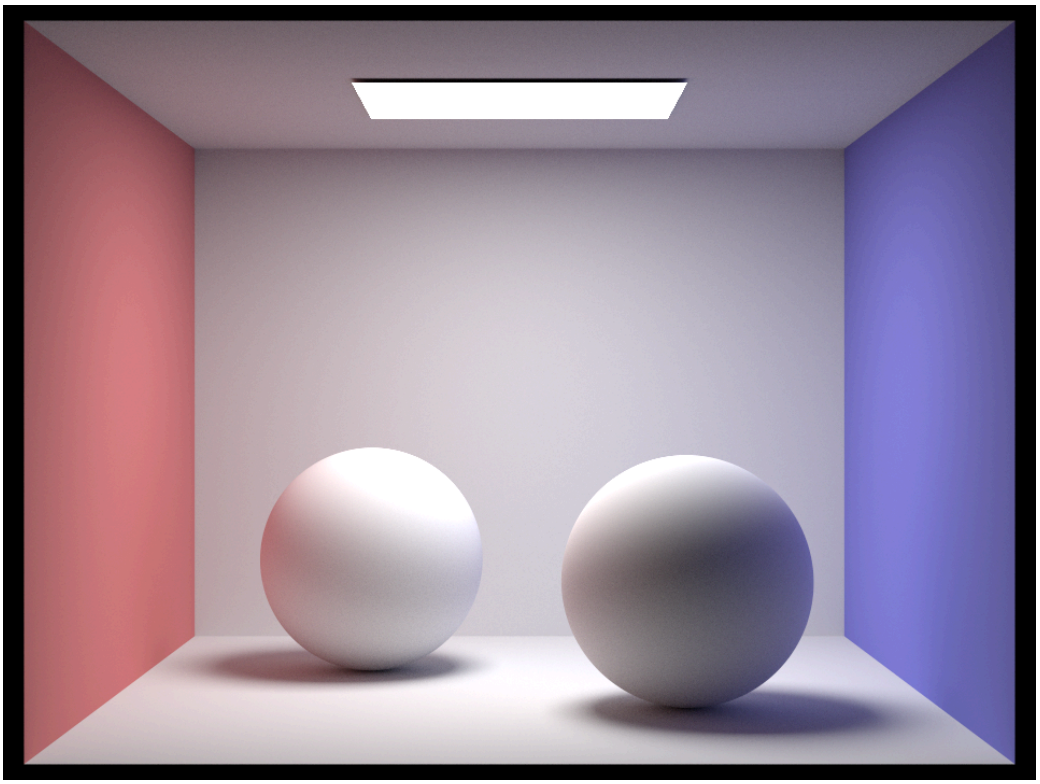
How was indirect lighting implemented?

Inside the `at_least_one_bounce_radiance(...)` function, I add the logic for accumulating indirect light bounces of length `N` paths. I first gather the radiance for a single bounce given the current ray and intersections, then I check whether we continue this recurssion based on the russian roulette rule. I set my probability of termination equal to `0.35` then call `coin_flip(...)` to test whether this computation continues. If it does, then I sample the bsdf to generate an outgoing direction, `wi`. Then, I create a new ray with origin `hit_p` and direction `wi`. Afterwards, I check whether this new ray intersects the `bvh` and if it does, then I recursively call `at_least_one_bounce_radiance(...)` to get the light bounces from the rest of the iterations.

Images rendered with global illumination



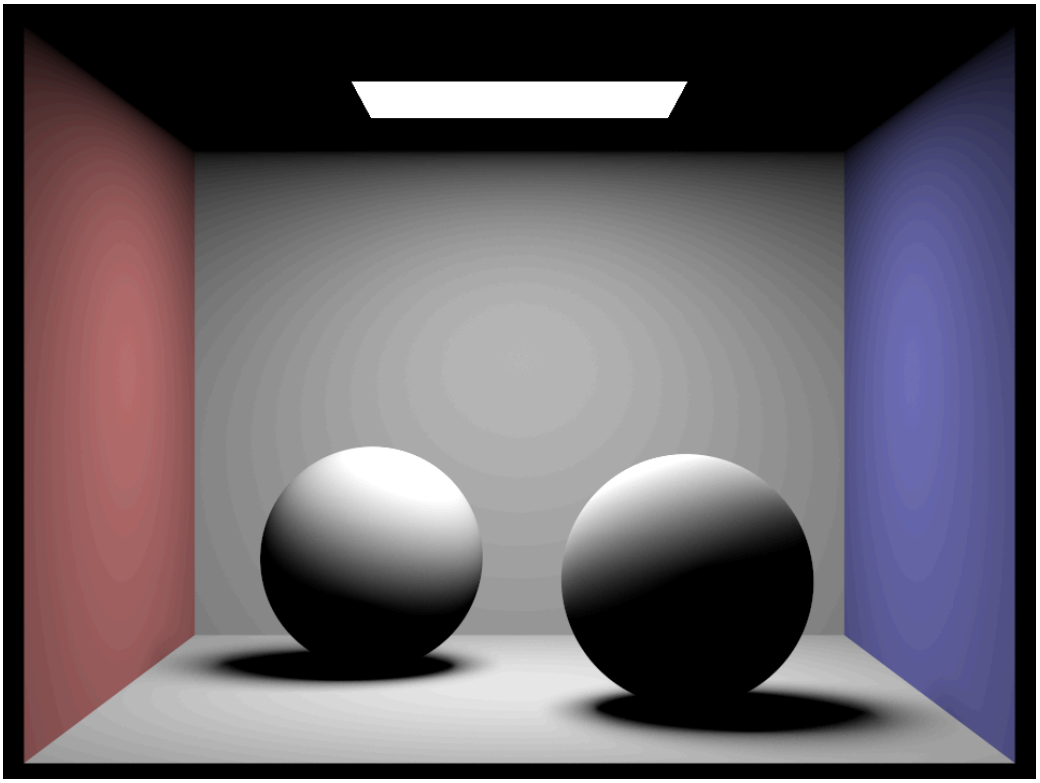
bunny.dae



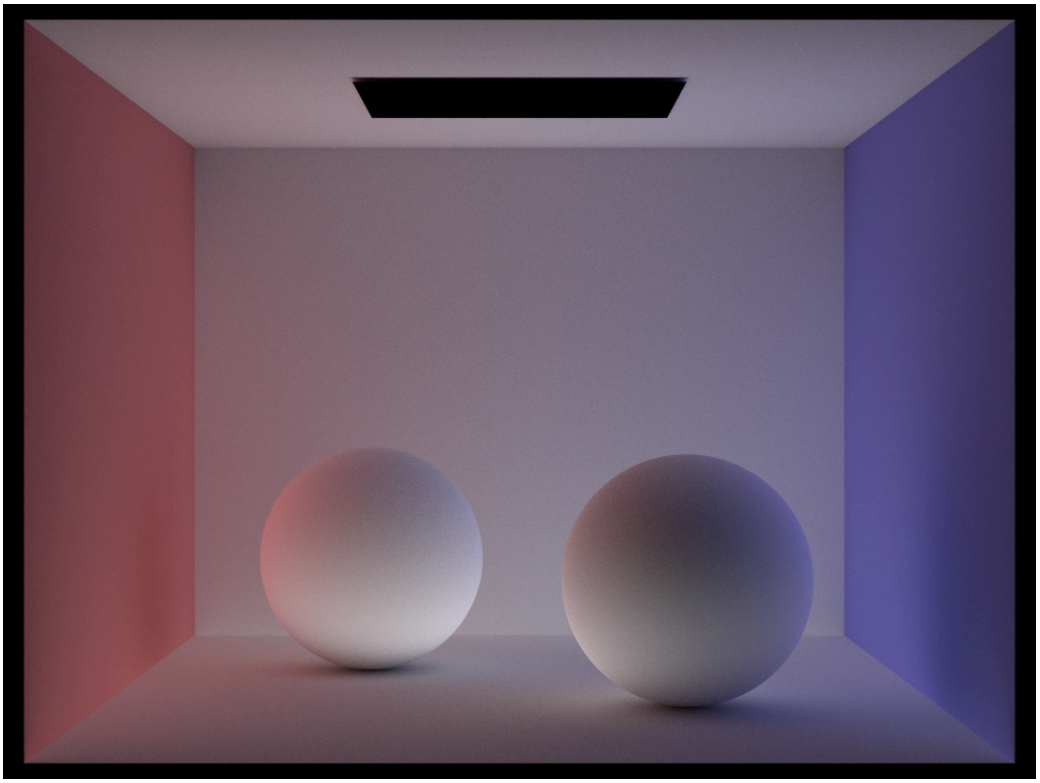
sheres.dae

How does only direct illumination compare to only indirect illumination?

I notice that direct illumination mostly addresses hard shadows, while indirect shadows addresses soft shadows. When looking at the left and right walls, I notice that shadows casted by the spheres is absent in the direct illumination render but present in the indirect illumination render. Additionally, I notice that the floor has a slight tint of color from the light that bounces off of the walls. I think this is a pretty unique and interesting feature about indirect lighting that really brings the image together.



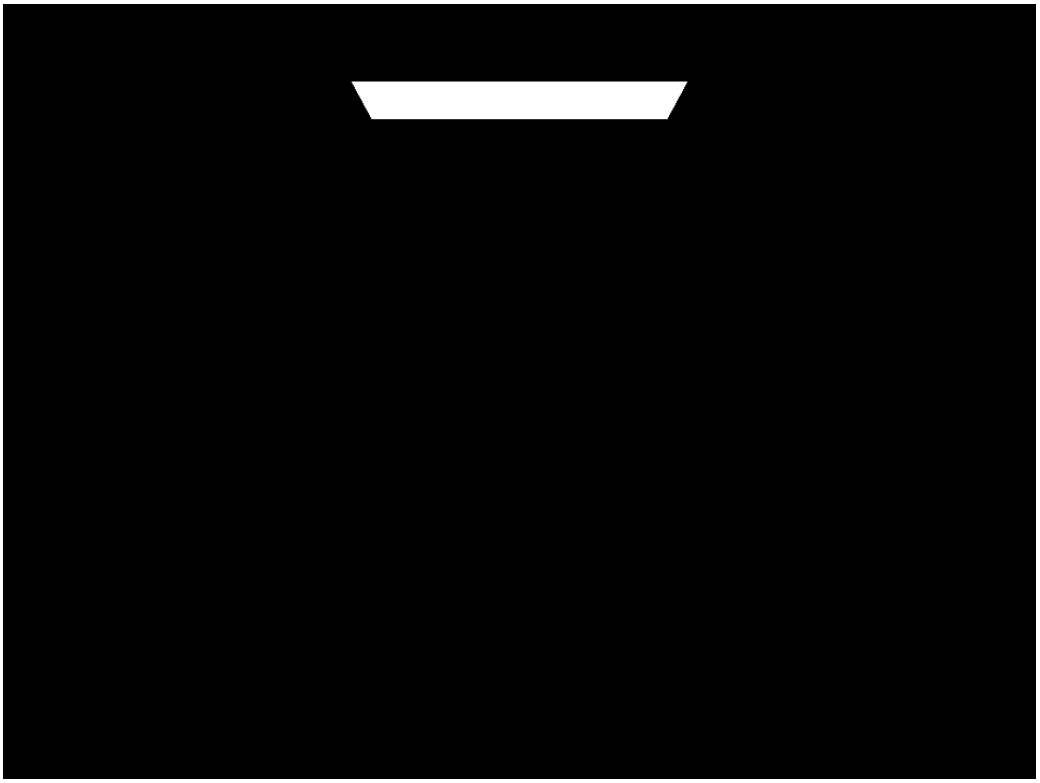
direct illumination only



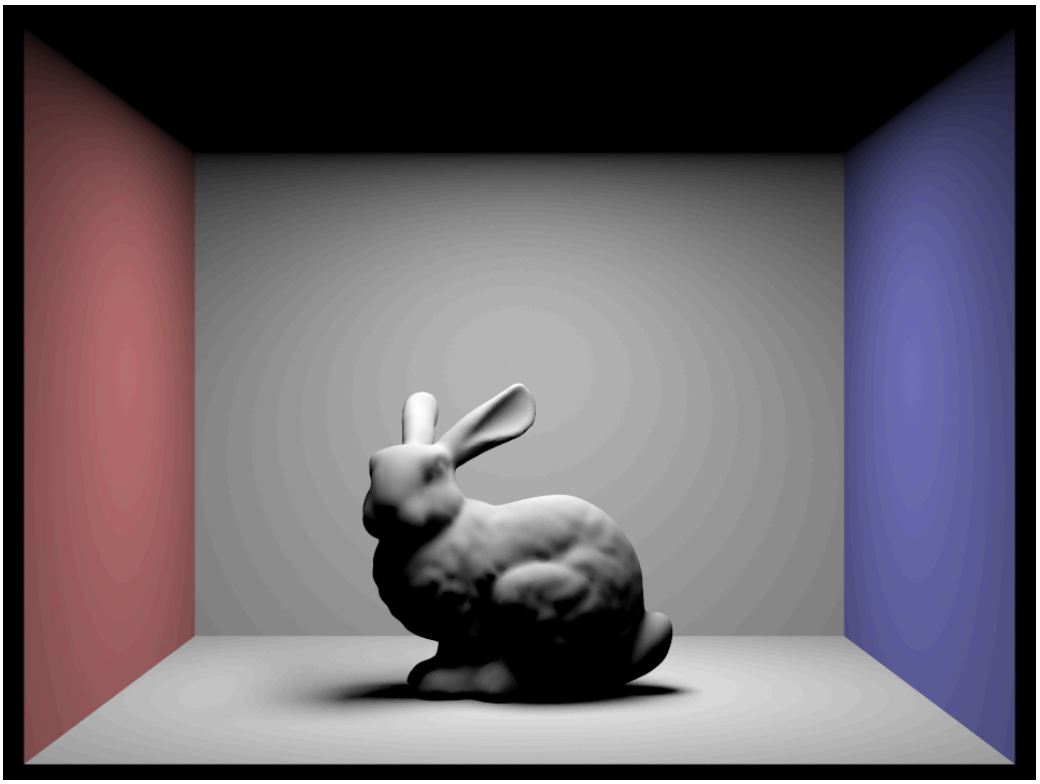
indirect illumination only

Render the mth light bounce. What happens for the 2nd and 3rd bounce of light? How does it contribute to the quality of the rendered image compared to rasterization?

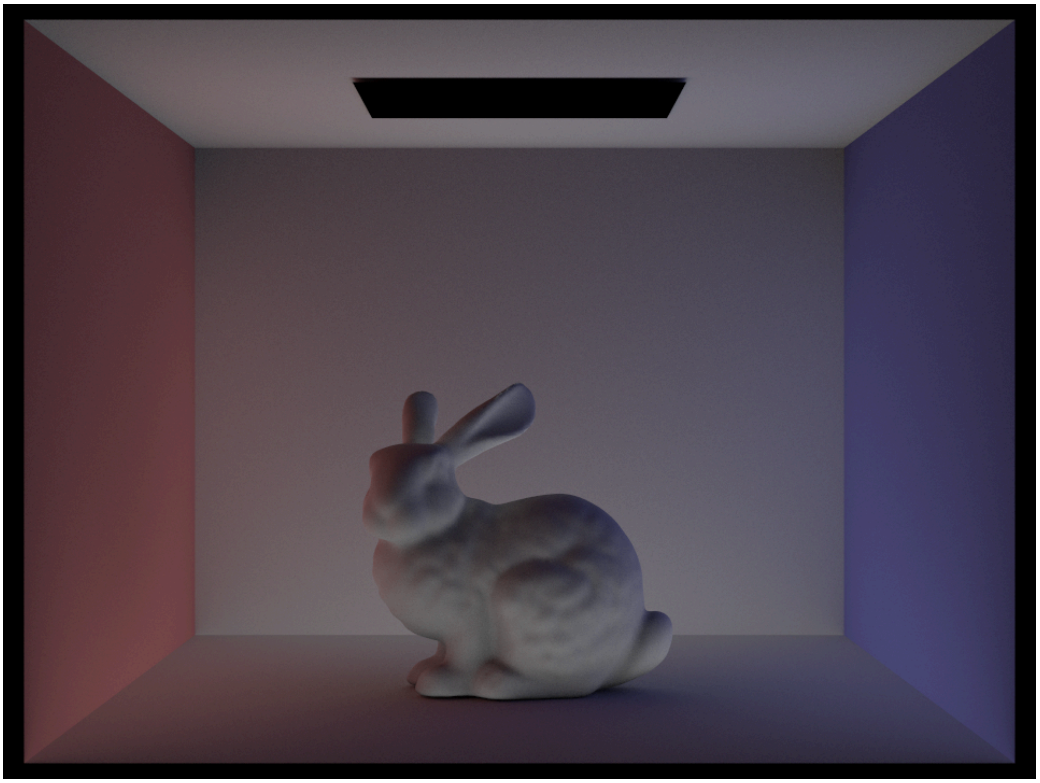
Between the 2nd and 3rd bounce renders, I notice that the image in general gets much darker, especially the bunny. This makes sense because by the 3rd bounce, most of the light has dispersed or been absorbed by the surrounding objects. In terms of the quality of the rendered image compared to the rasterization, I'd say that it adds a lot of definition to the render. Soft shadows are defined and color is diffused in a way that makes the render look more like real life.



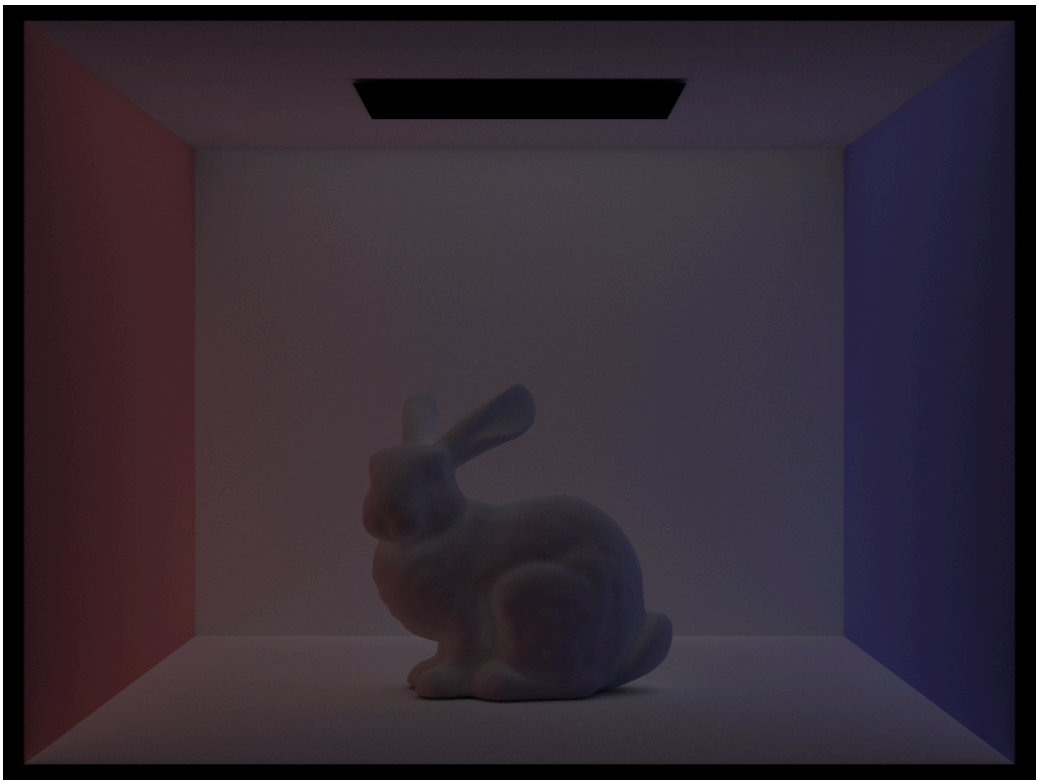
0th bounce



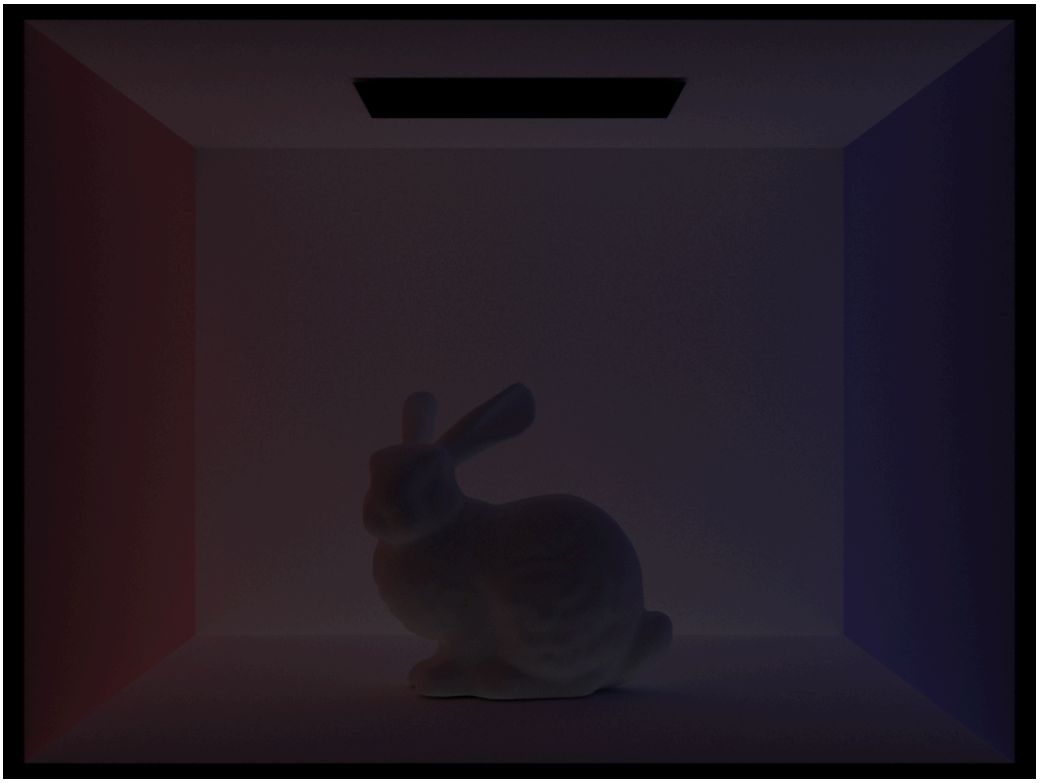
1st bounce



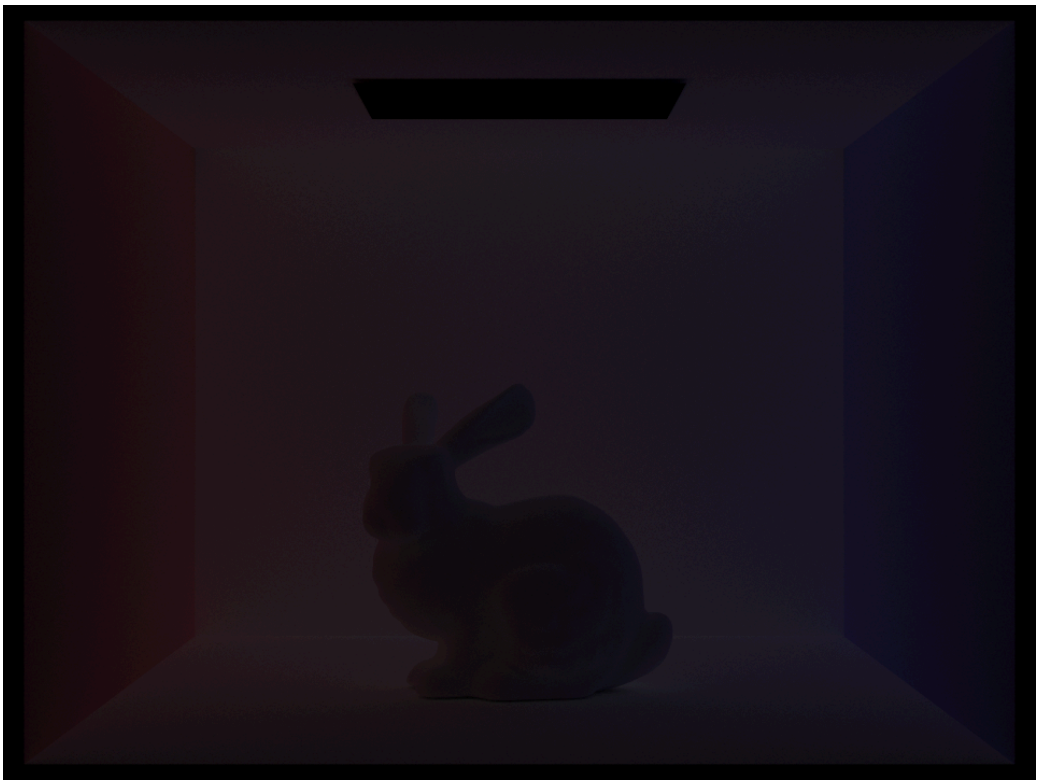
2nd bounce



3rd bounce



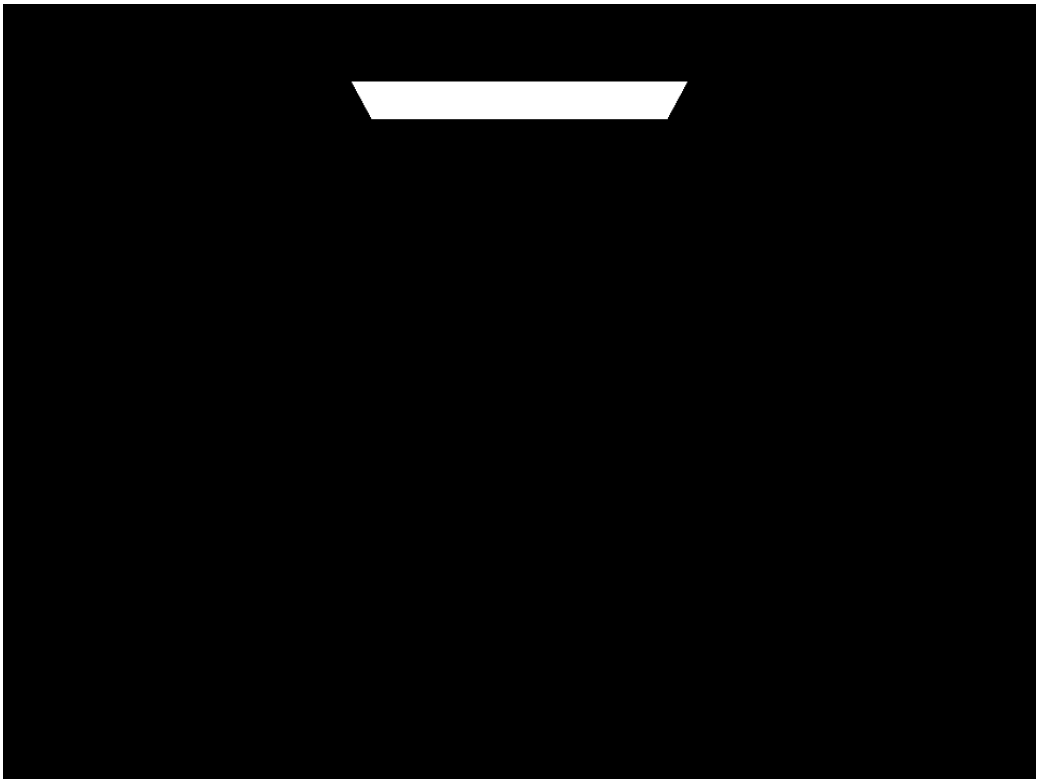
4th bounce



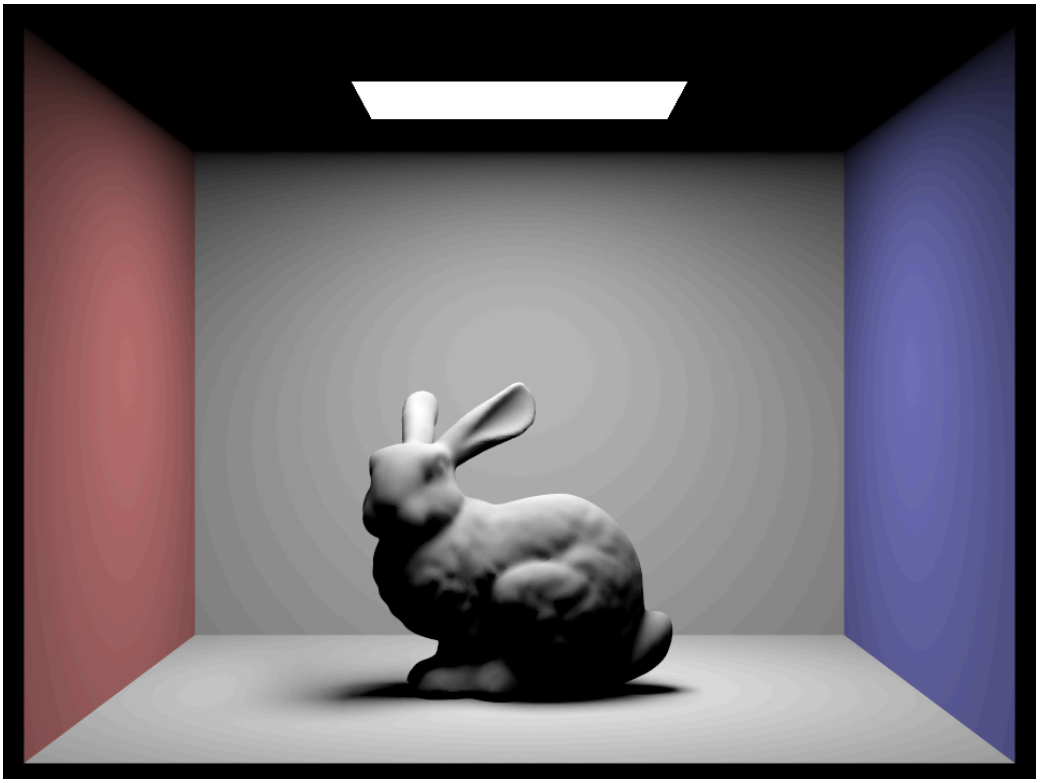
5th bounce

How do rendered images look with different ray depths? (without Russian Roulette)

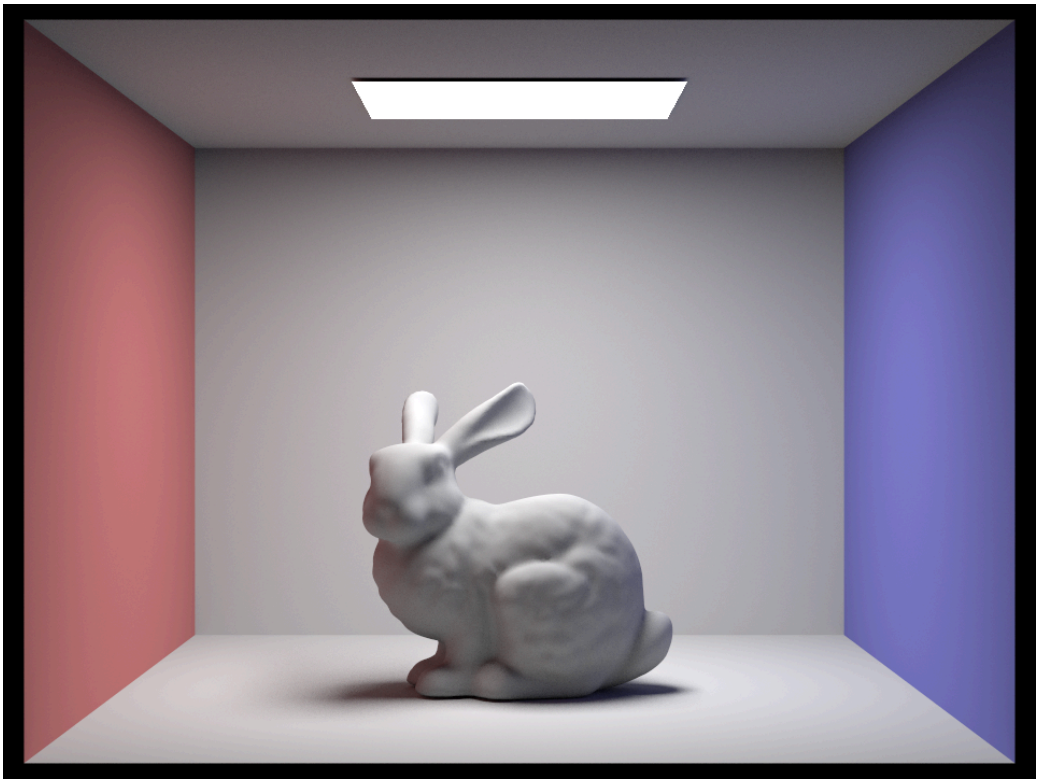
Images look more "real" with increased ray-depth. Especially in the range of 1-3 ray depths, I think this is the range where change is most noticeable and most impactful to the render. Ray depths after 3 seem to make little to no noticable impact on how a render looks to the naked eye.I notice that the major difference between 2 ray depths and 5 ray depths is the amount of light diffusion. On the bunny and on the walls, the light looks to be more diffused; the white shades on the bunny looks more natural and warmer than the white with 2 ray depths.



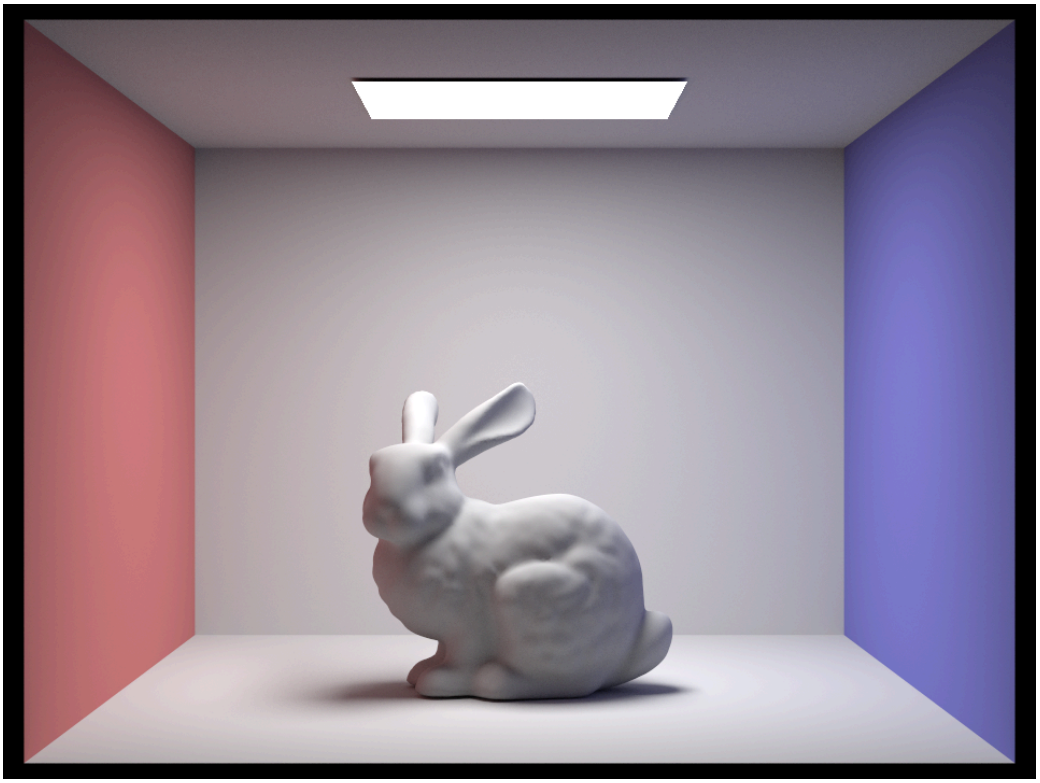
ray depth 0



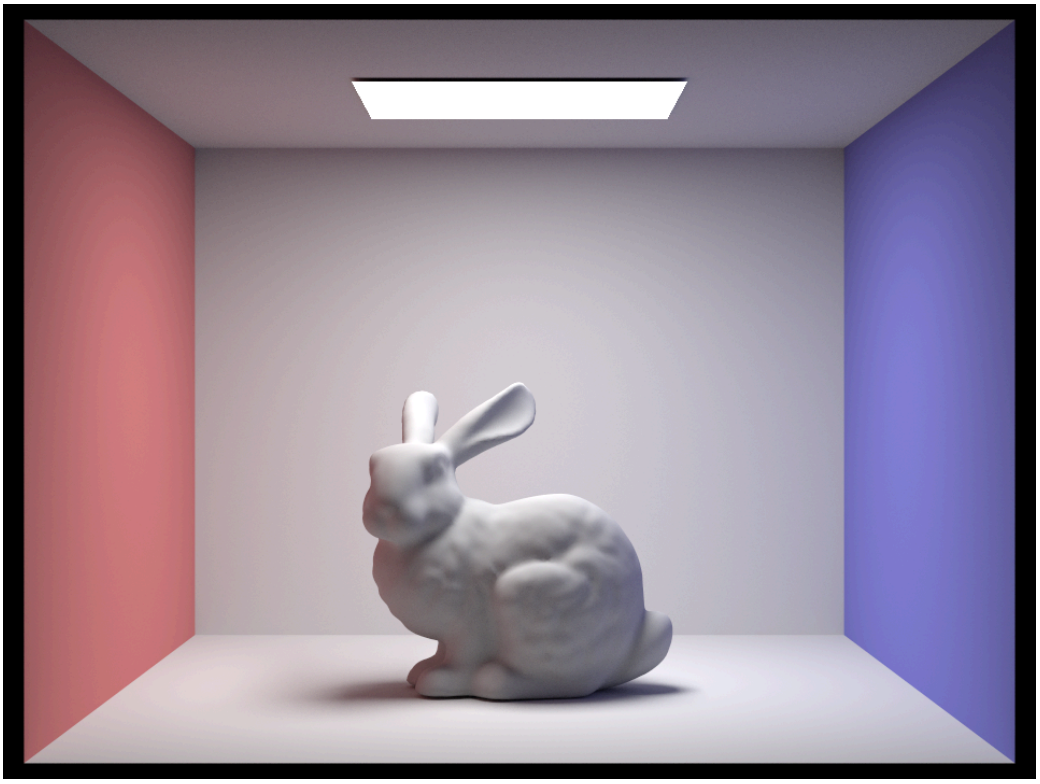
ray depth 1



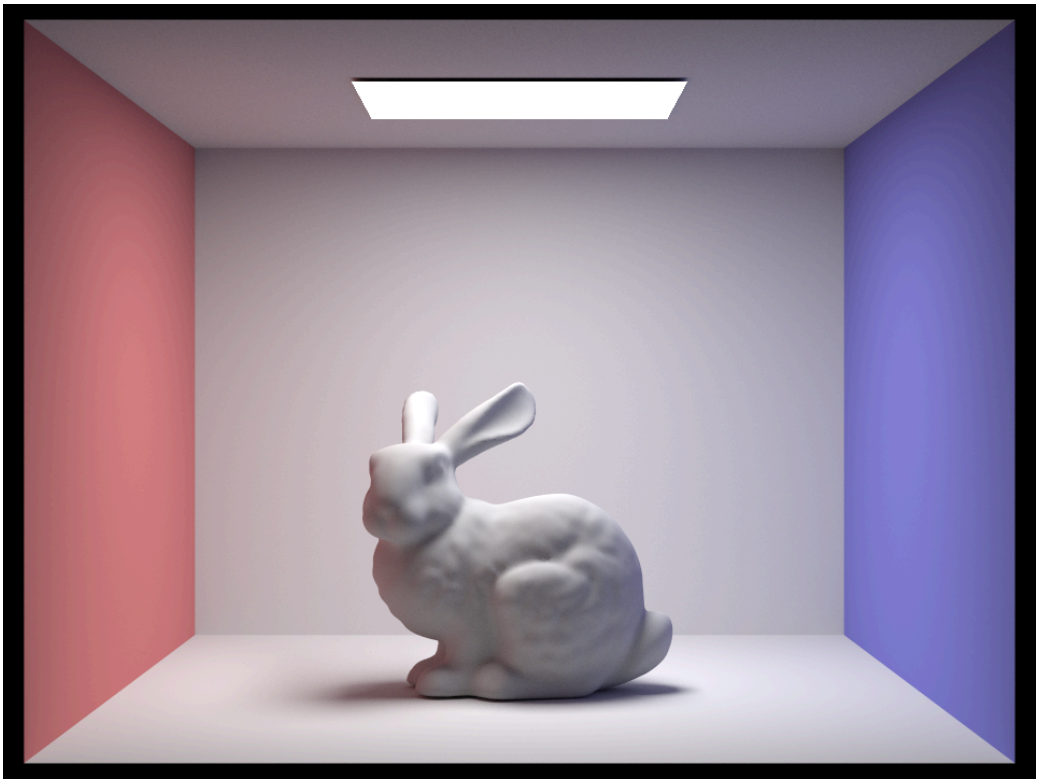
ray depth 2



ray depth 3



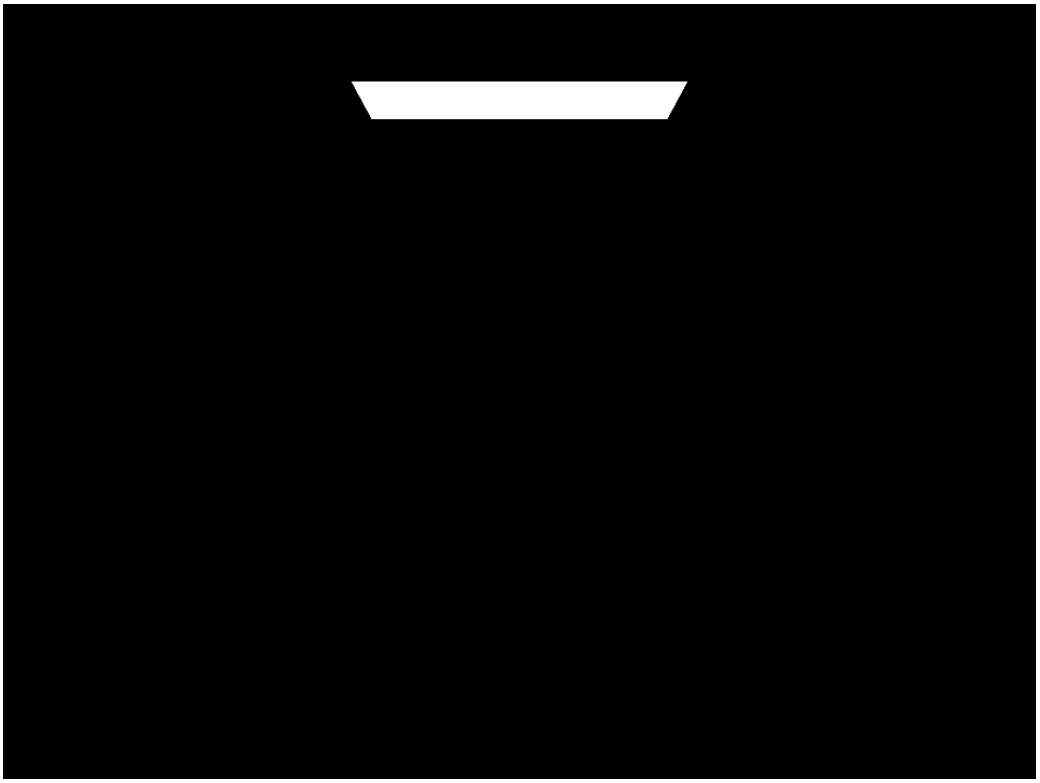
ray depth 4



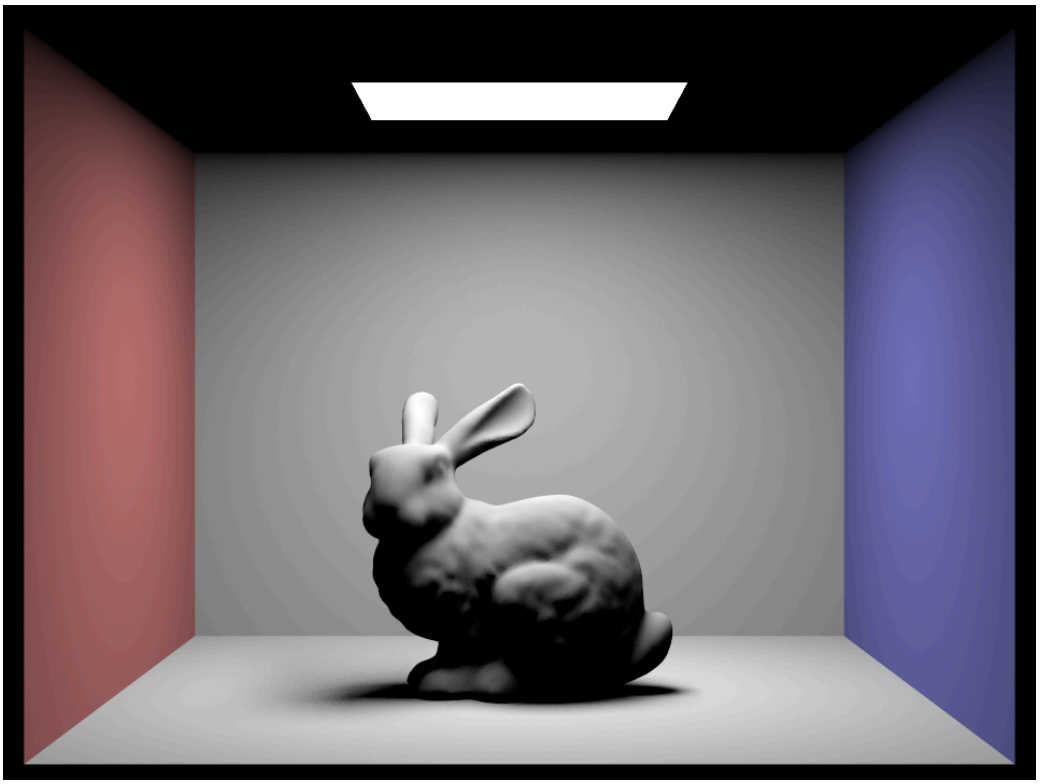
ray depth 5

How do rendered images look with different ray depths? (with Russian Roulette)

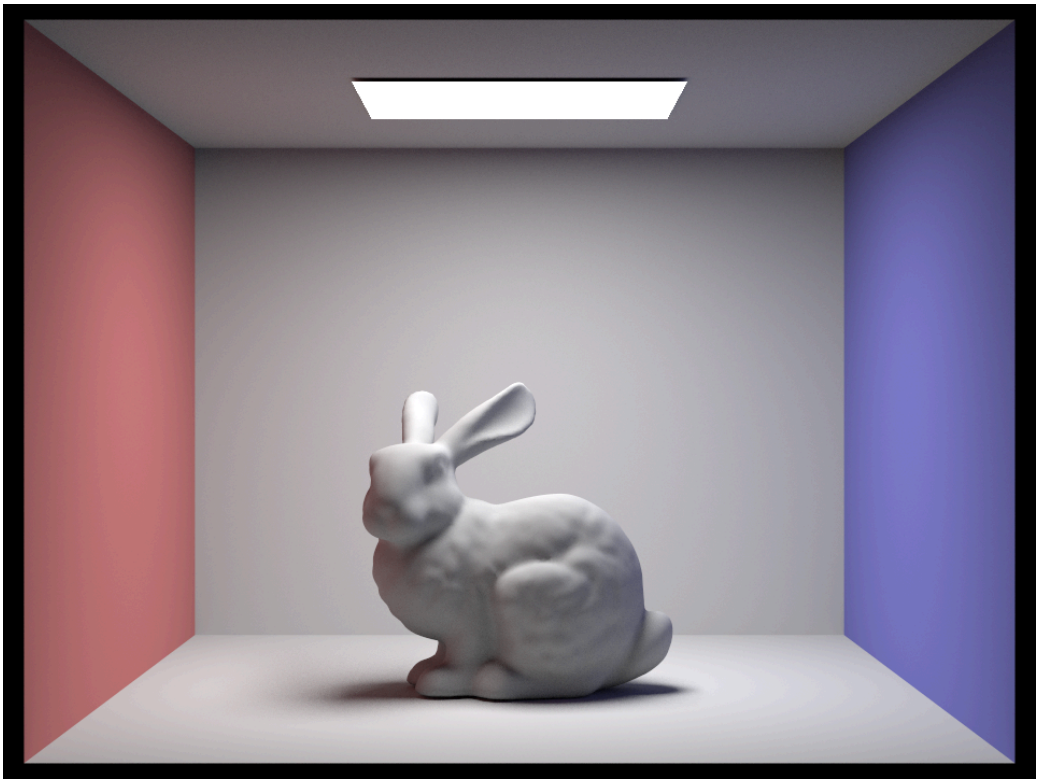
Similar to the rendered images without Russian Roulette, I'd say that the images with increased ray depth look more "real". Especially when bumping up the number of rays to 100, I'd say that the rendered image looks very nice. This type of convergence and unbias result is something that Russian Roulette helps up get.



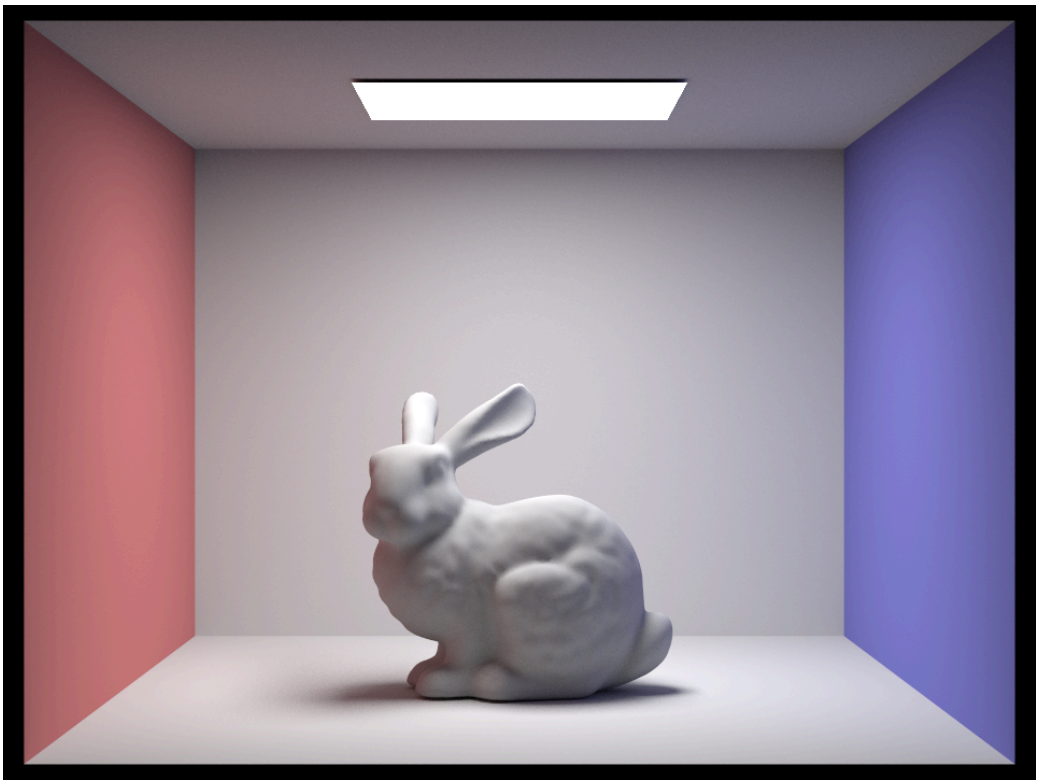
ray depth 0



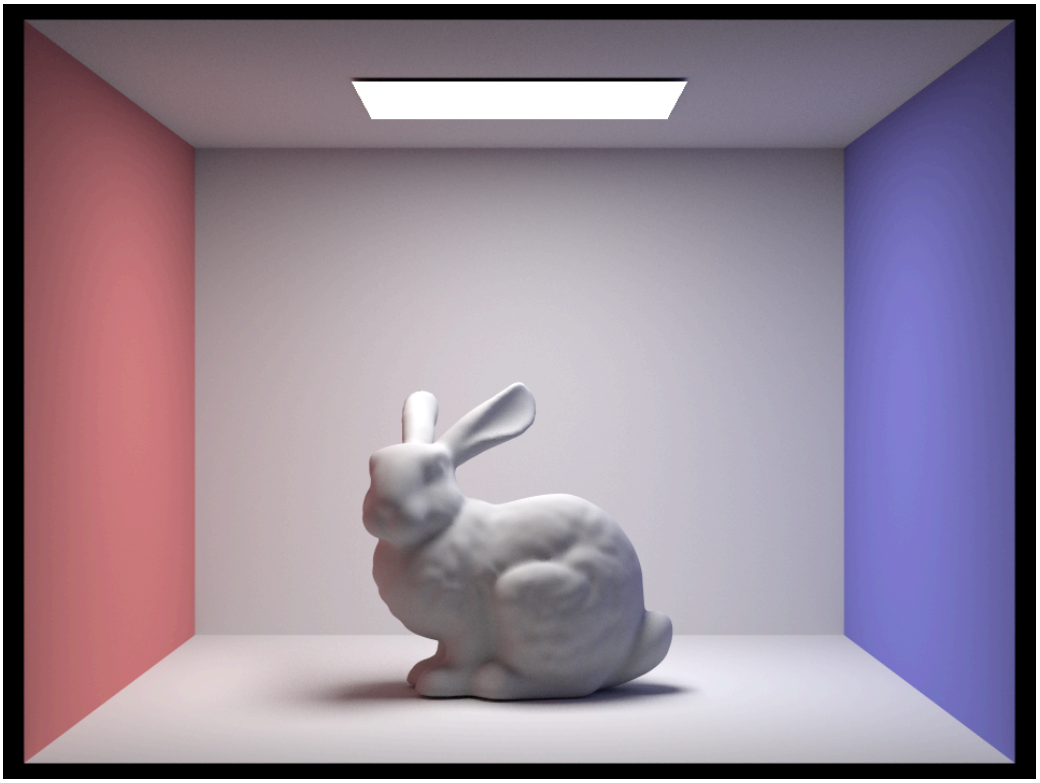
ray depth 1



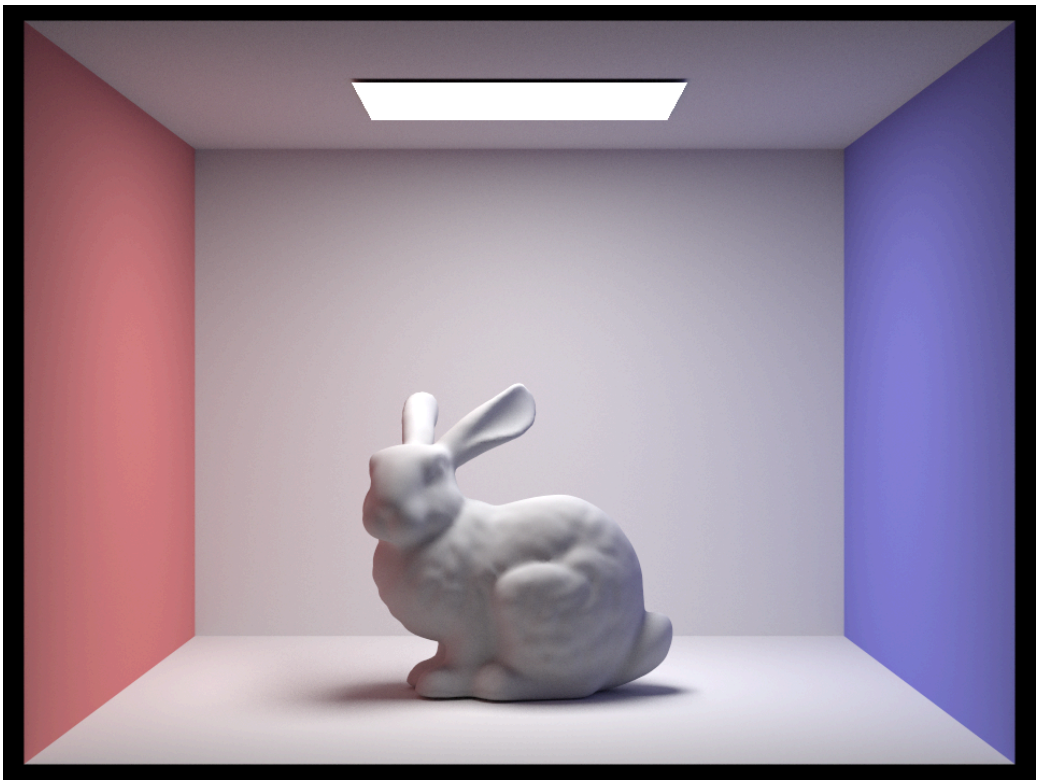
ray depth 2



ray depth 3



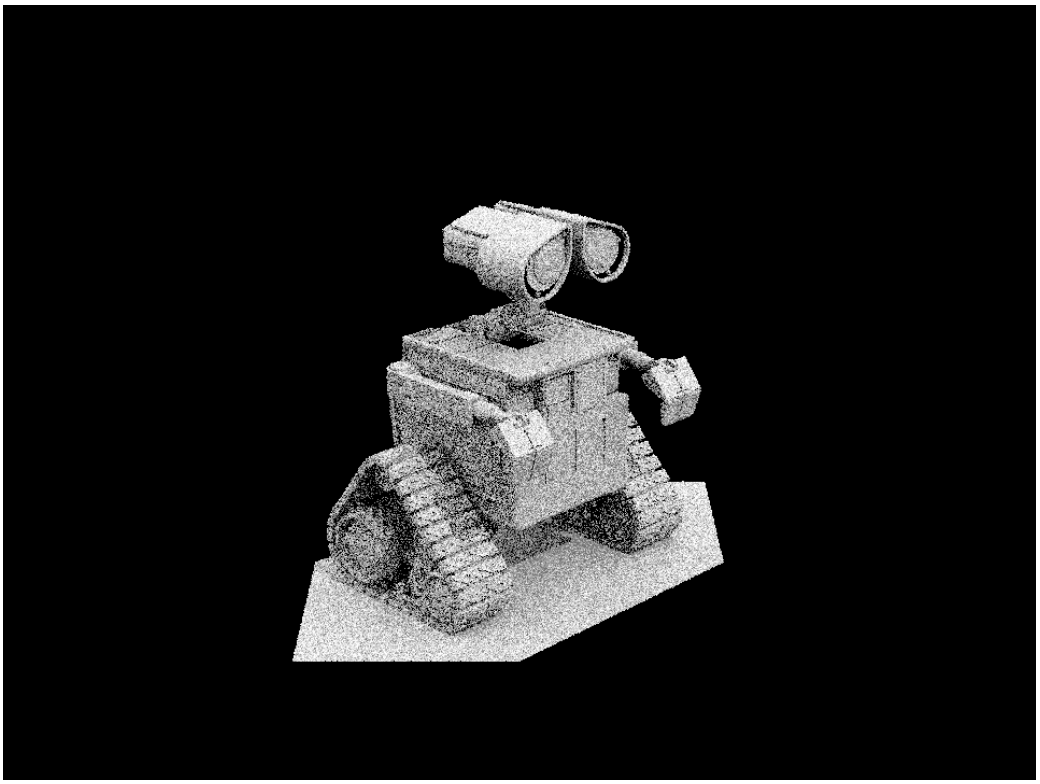
ray depth 4



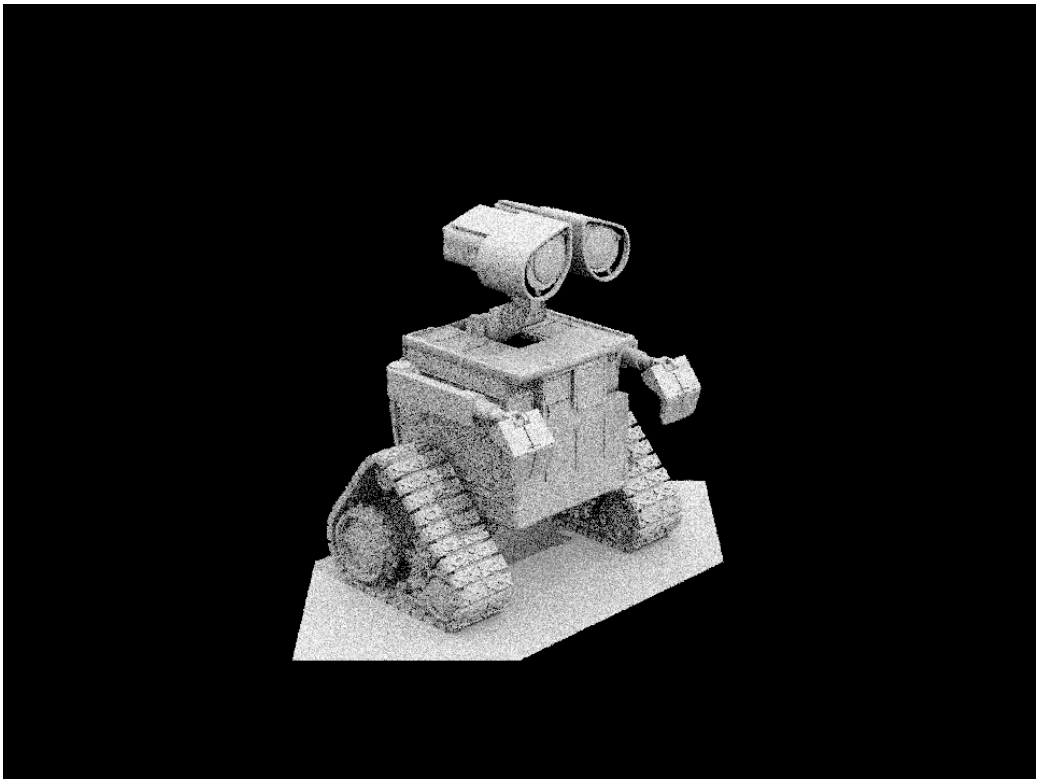
ray depth 100

How do rendered scenes compare with different sample-per-pixel rates?

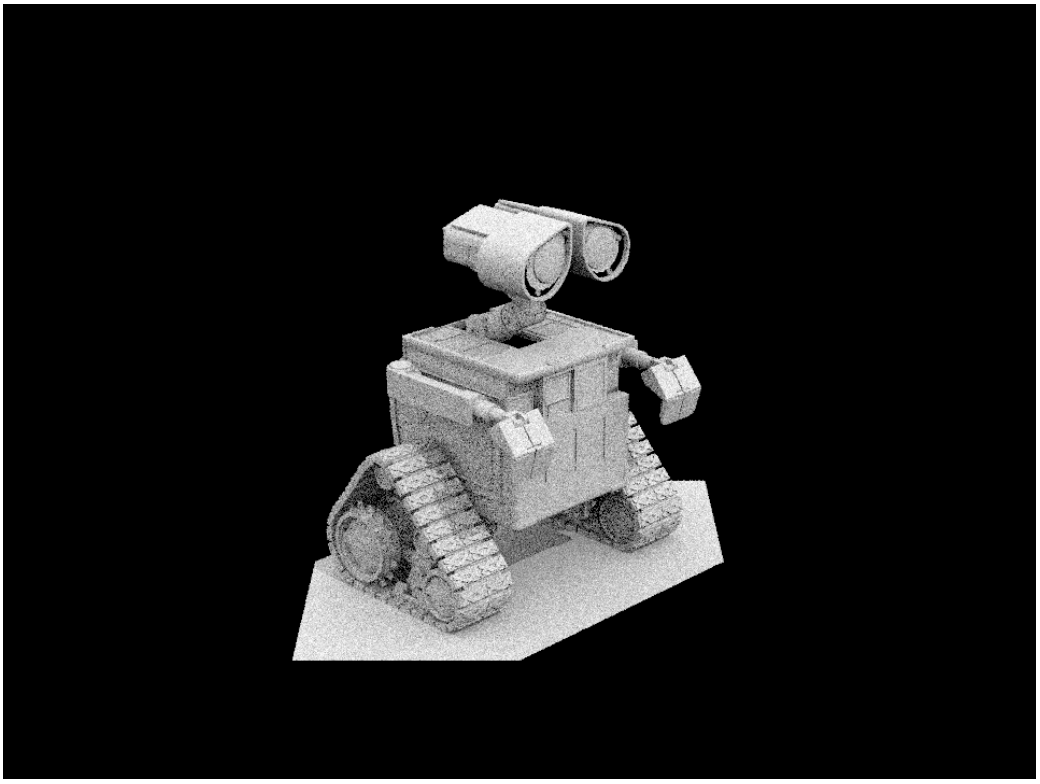
In general, as scenes are rendered with higher sample-per-pixel rates, the more clear and defined they look. For example, the image rendered with 1 sample per pixel looks grainy and noisy. On the other hand, the image rendered with 1024 samples per pixel looks basically flawless. One can point out the details on wall-e's gears, joints, and caterpillar track.



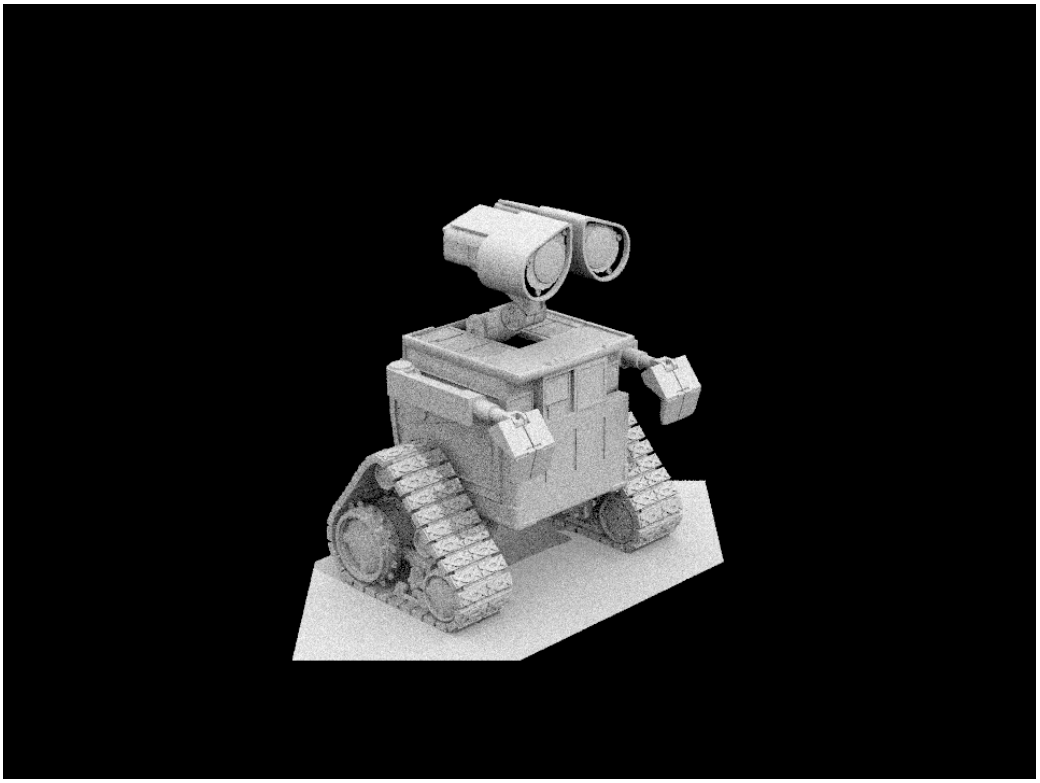
1 sample per pixel



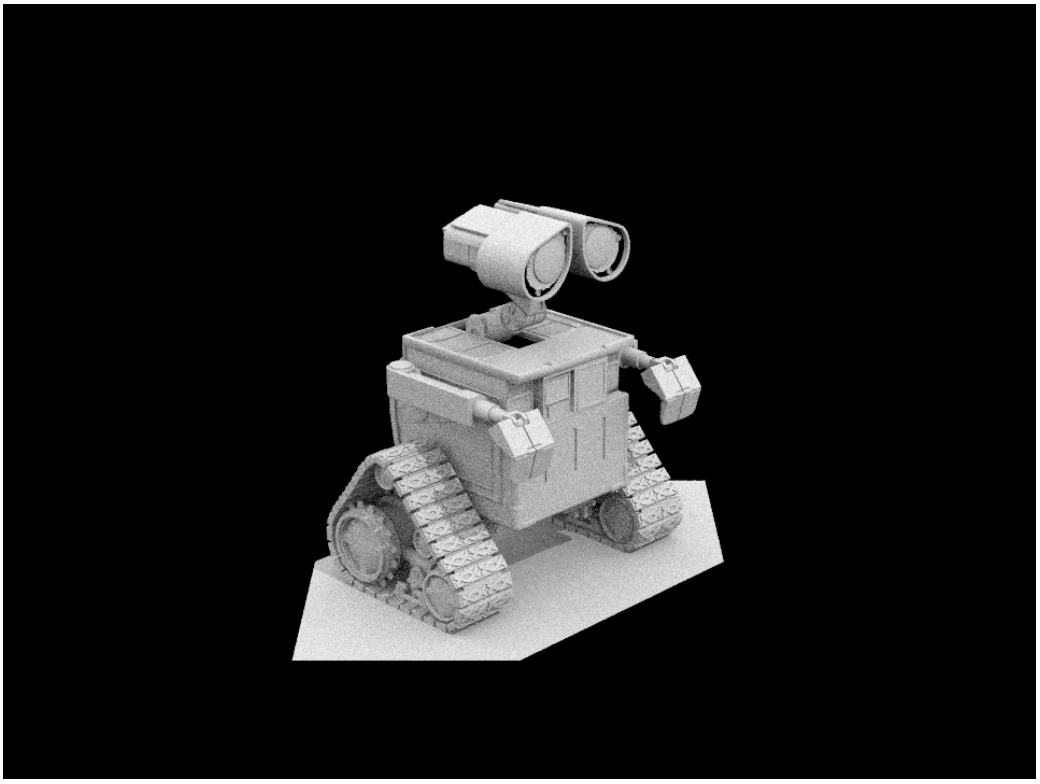
2 samples per pixel



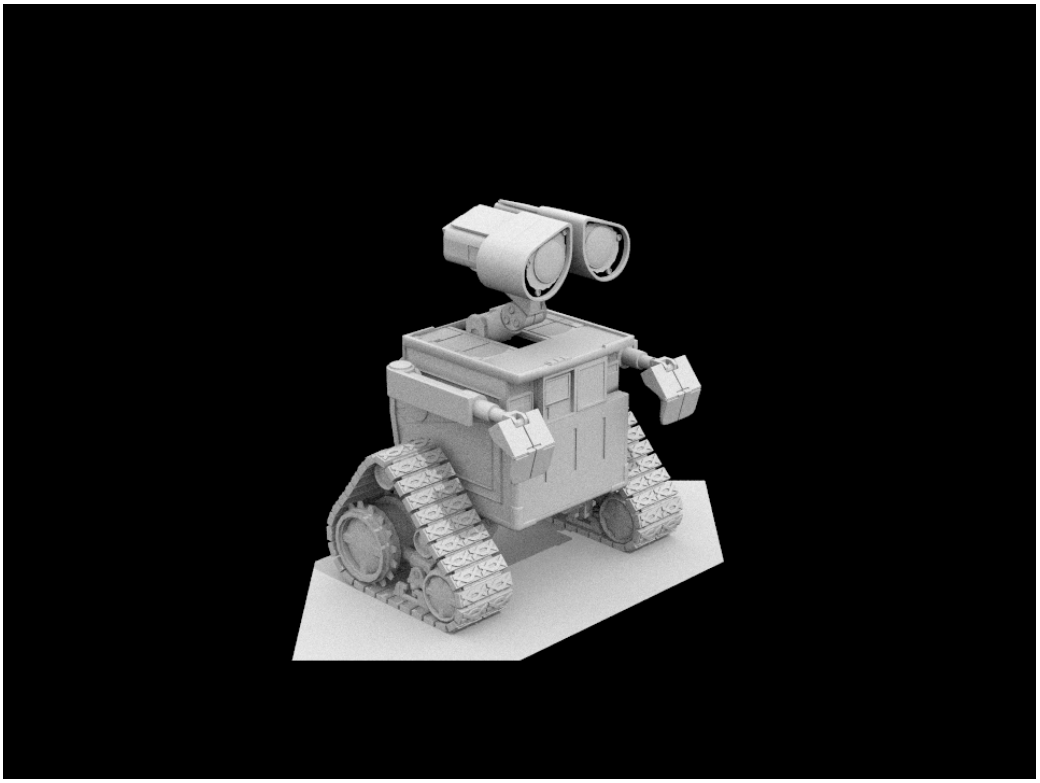
4 samples per pixel



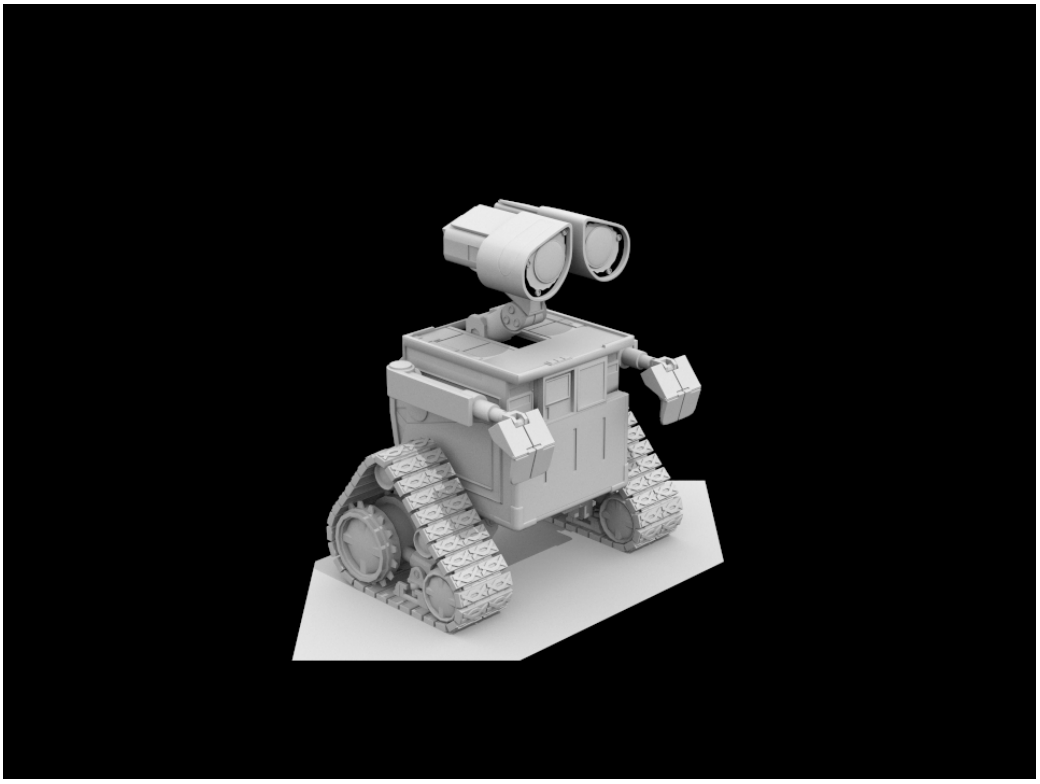
8 samples per pixel



16 samples per pixel



64 samples per pixel



1024 samples per pixel

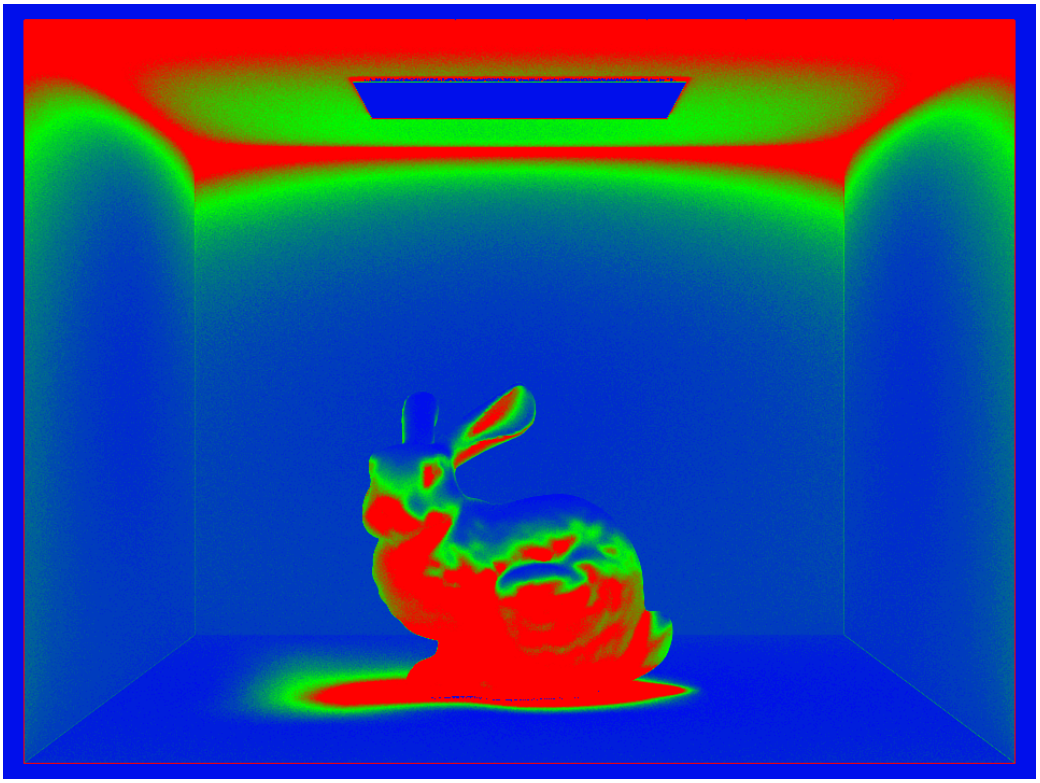
Part 5: Adaptive Sampling

What is adaptive sampling?

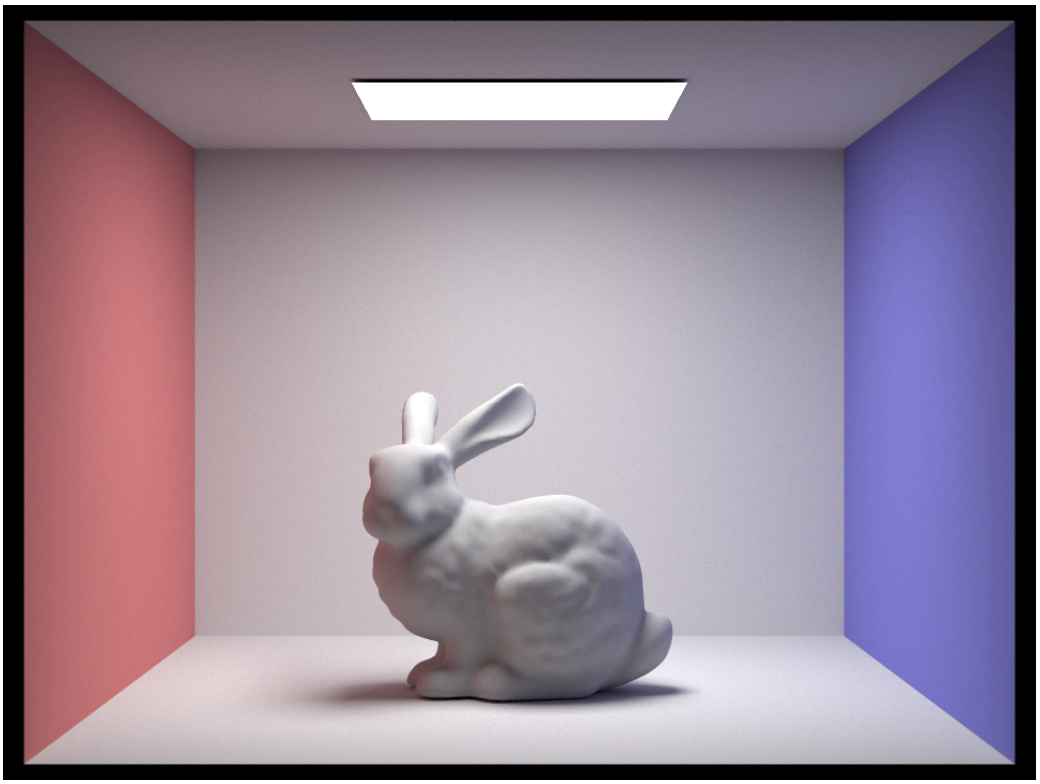
Adaptive sampling is a technique used to improve the efficiency of rendering scenes by dynamically changing the sampling rate of pixels based on how noisy they are. Because some pixels converge faster with low sampling rates, we have some computation time by not sampling points more than we need to. Therefore, we can focus on sampling pixels that are more noisy.

How was adaptive sampling implemented?

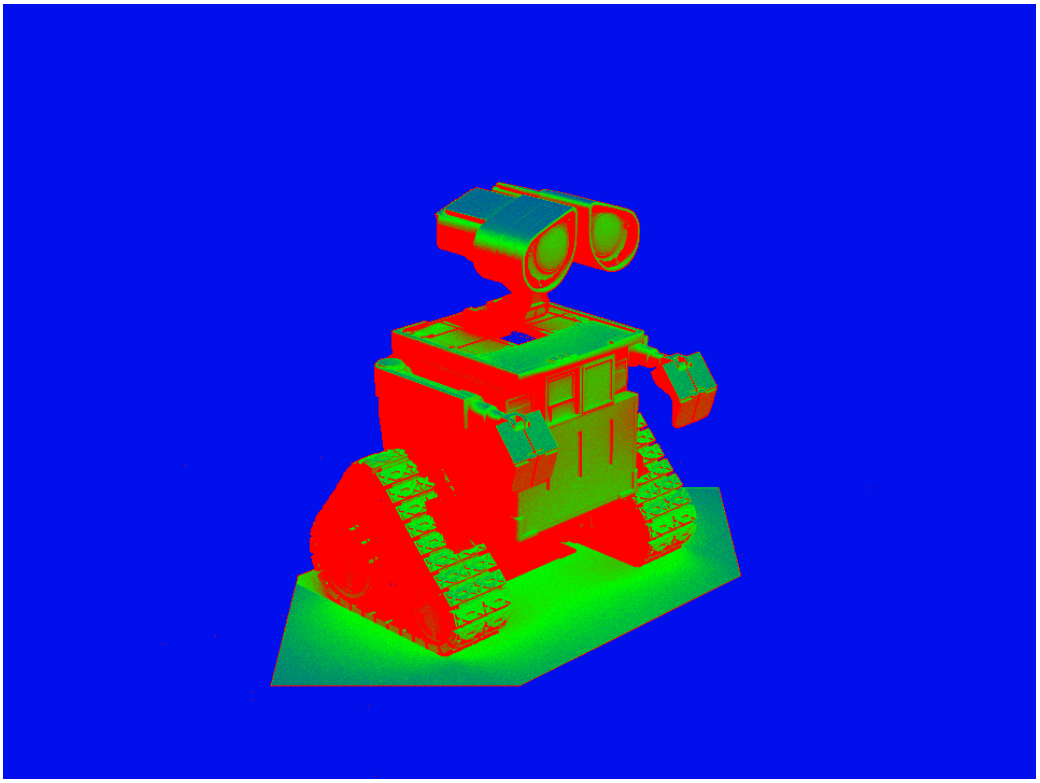
By altering the `raytrace_pixel(...)` function, I was able to keep track of variables `s1` and `s2` which represent the sum of sample illuminations and the sum of squared sample illuminations, respectively. These are then used to compute sample mean and variance every `samplesPerBatch` times and compute whether this pixel has converged or not. If a pixel has converged, then we stop sampling, else, we continue until we converge or reach the maximum number of samples.



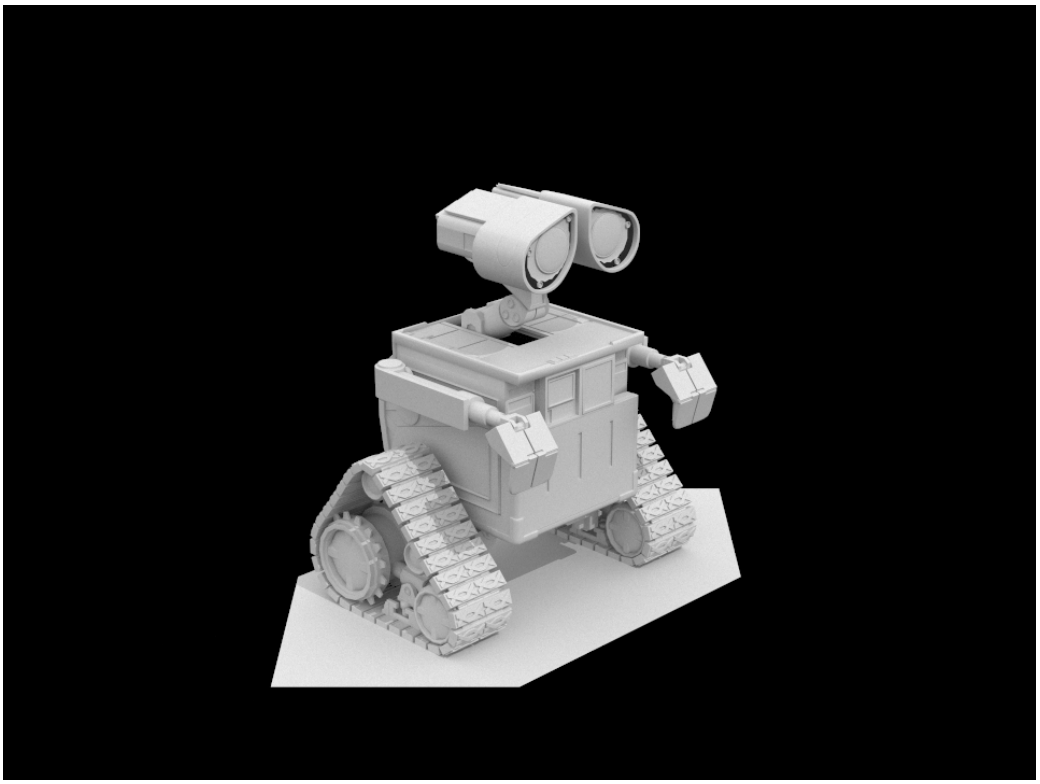
depiction of sampling rate



final render



depictions of sampling rate



final render

Emmanuel Cobian Duarte - CS 184 - Spring 2024