CS 184: Computer Graphics and Imaging, Spring 2024

Project 3: Path Tracer

Satvik Muddana

Link to my writeup webpage.

Overview

The overview of this project was making a ray tracer that started off as a basic implementation of direct lighting and the hemisphere/importance methods. After several optimizations such as indirect lighting, russian roulette, and adaptive sampling, my ray tracer reached a point where it can render a lot of .dae files well. A lot of analysis was also done on the difference in lighting methods and optimizations.

Part 1: Ray Generation and Scene Intersection (20 Points)

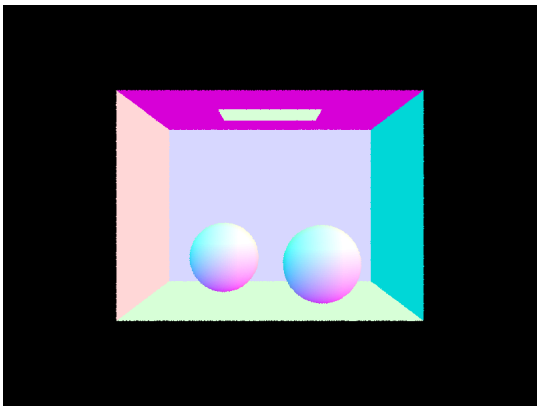
Walk through the ray generation and primitive intersection parts of the rendering pipeline.

My implementation for ray generation starts by scaling the axes of the pixel coordinate that is given to coordinate frame that is normalized. I do this in the same calculation as the next step which is taking this coordinate frame and using scaling/translating perpendicular to the Z-axis of the camera to map it to the image frame field of view. This is let us know to generate a ray from the camera. Thus we can finally use the origin of the camera and the direction vector to create a ray. I did the primitive intersection part of the pipeline with the general idea of finding t-values and checking if they were in the primitive. The sphere was a bit more complicated than the triangle because it required quadratic equations and roots which might not always be a real solution.

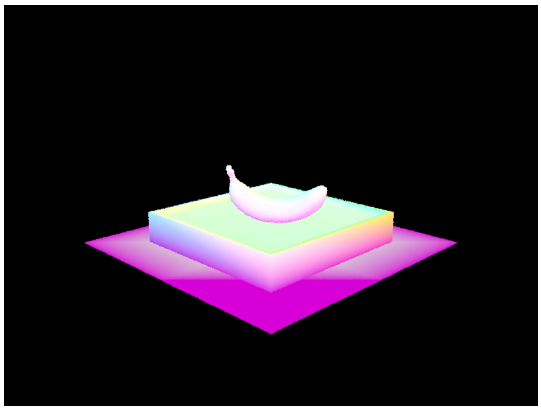
Explain the triangle intersection algorithm you implemented in your own words.

I implemented triangle intersection utilizing the Moller-Trumbore Algorithm, an approach that blends Barycentric coordinates with an implicit plane definition. These are used to decide whether the specific point is in the triangle intersections. By meticulously ensuring that all Barycentric Coordinates fall within the prescribed range of [0,1] and verifying the intersection's t-value against the scene's maximum and minimum bounds, I can verify intersections with triangle primitives.

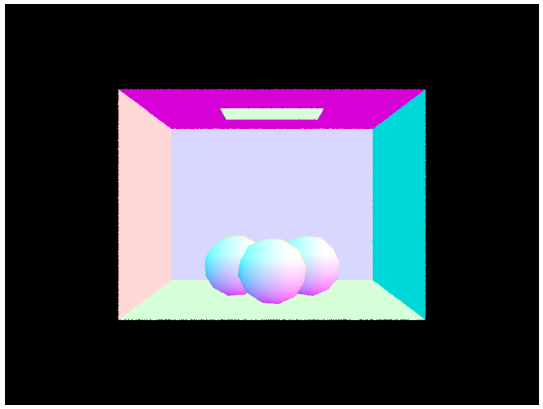
Show images with normal shading for a few small .dae files.

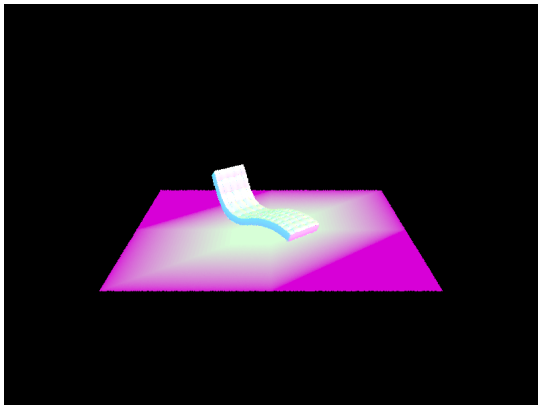


CBspheres.dae



banana.dae


CBgems.dae


bench.dae

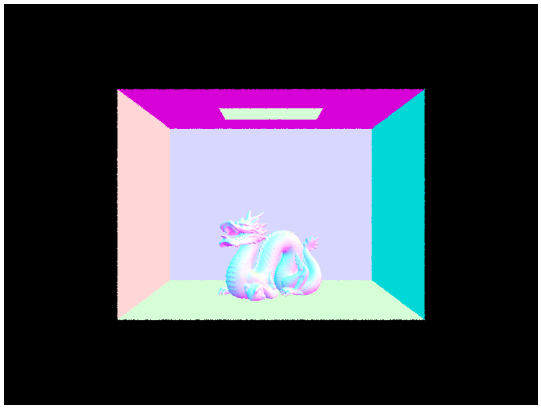
Part 2: Bounding Volume Hierarchy (20 Points)

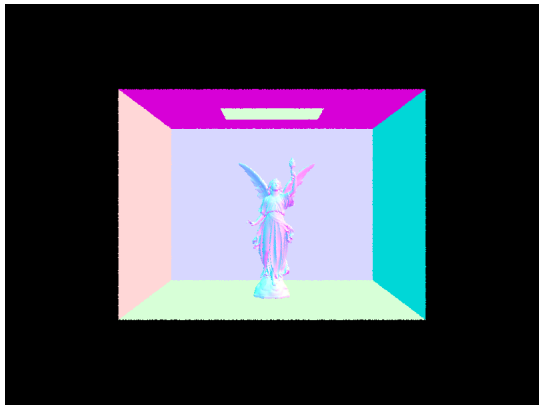
Walk through your BVH construction algorithm. Explain the heuristic you chose for picking the splitting point.

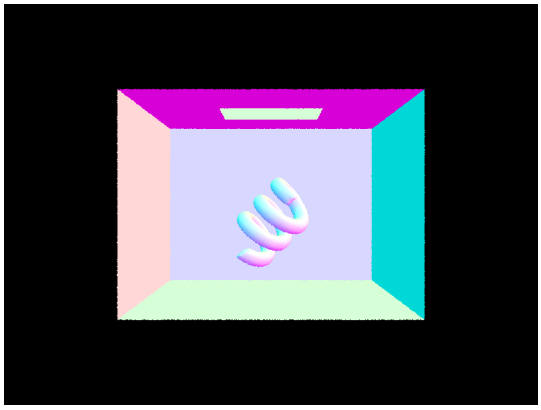
Initially, for a given BVH node, I calculated the centroid positions of all primitives encapsulated within the bounding box of the node. This computation yielded three potential split points along the x, y, or z axis. Next, I determined the cost associated with splitting each axis at its respective mean centroid coordinate. The cost function was straightforward, computed as the total surface area of the bounding boxes of the resultant child nodes from the split point, multiplied by the number of primitives in each child node. Finally, I chose the split point that incurred the lowest cost. I used mean for the heuristic because it was simply easier to calculate versus having to sort steps to find something like the median.

Show images with normal shading for a few large .dae files that you can only render with BVH acceleration.


cow.dae


CBdragon.dae


CBlucy.dae


CBcoil.dae

Compare rendering times on a few scenes with moderately complex geometries with and without BVH acceleration. Present your results in a one-paragraph analysis.

Without the BVH the cow took 40.021 seconds to render. With BVH implemented, it took 0.1335 seconds. This a surprising amount of difference, which highlights the effect of BVH and its speed. It's time complexity turns O(n) to O(log(n)). With the cow image, it is easy

to visualize significance that a small change in time complexity can have.

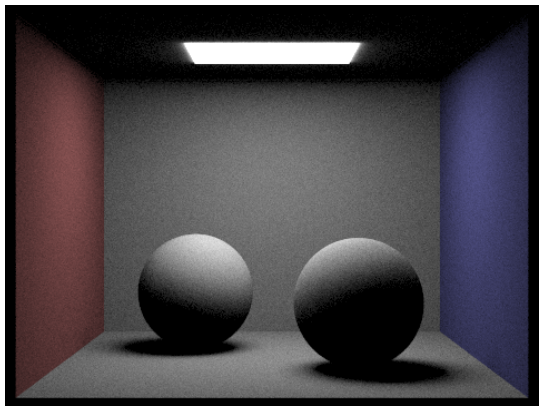
Part 3: Direct Illumination (20 Points)

Walk through both implementations of the direct lighting function.

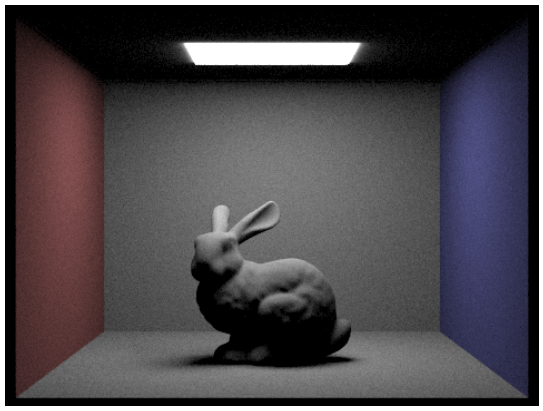
For both Uniform Hemisphere Sampling and Importance Sampling, the idea is to illuminate the image plane with rays originating from a light source after one bounce. Both implementations involve sampling ray directions for approximating the reflection equation and checking if they intersect the light source. Whenever an object intersects a ray's path, this implementation uses the Bidirectional Radiance Distribution Function (for diffuse surfaces) of the object's surface and the arriving irradiance to calculate the radiance that should be observed at that intersection point. The two implementations direct lighting differ in how they sample. In Uniform Hemisphere Sampling the rays are not necessarily heading towards a light source versus importance sampling which is guranteed. In imporantce sampling, a ray is only ignored if something obstructs it, so its irradiance is more accurate.

Show some images rendered with both implementations of the direct lighting function.

Uniform Hemisphere Sampling

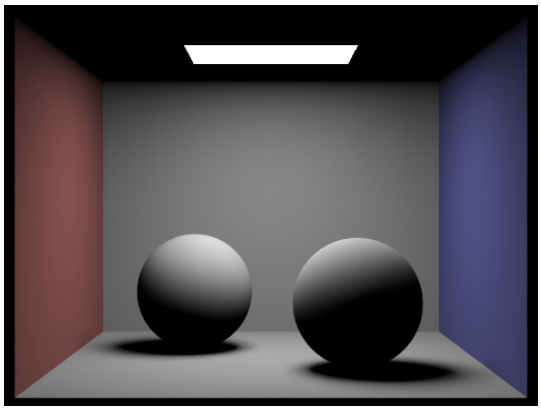


CBspheres_lambertian.dae

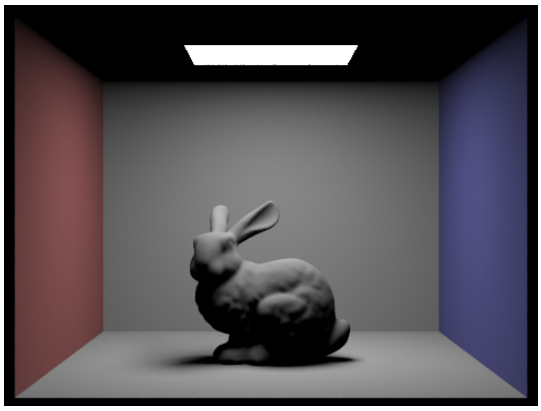


CBbunny.dae

Light Sampling

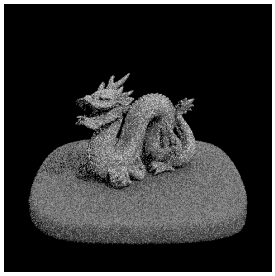


CBspheres_lambertian.dae



CBbunny.dae

Focus on one particular scene with at least one area light and compare the noise levels in soft shadows when rendering with 1, 4, 16, and 64 light rays (the -l flag) and with 1 sample per pixel (the -s flag) using light sampling, not uniform hemisphere sampling.



1 Light Ray (dragon.dae)

4 Light Rays (dragon.dae)


16 Light Rays (dragon.dae)


64 Light Rays (dragon.dae)

The effect is very clear after changing just one parameter, light rays. As the number of light samples increased, the visible noise decreased.

Compare the results between uniform hemisphere sampling and lighting sampling in a one-paragraph analysis.

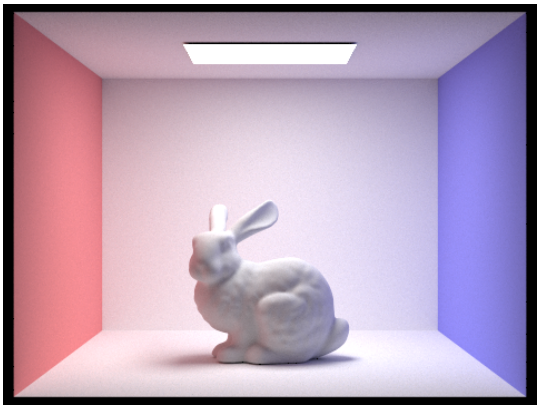
When comparing uniform hemisphere sampling and lighting sampling, it is evident that lighting/importance sampling generates an image with significantly less noise than uniform hemisphere sampling. The larger amount of rays contributing to nothing in uniform hemisphere sampling can be witnessed in the noisiness of the rendered images.

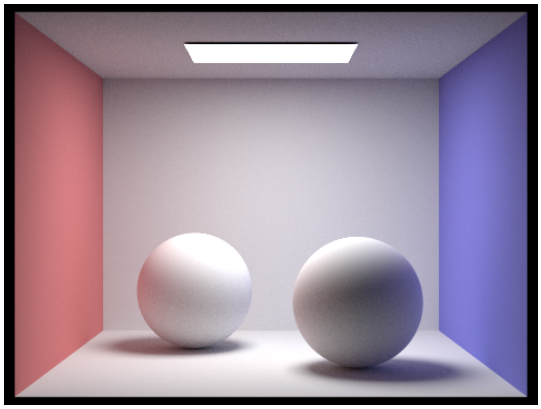
Part 4: Global Illumination (20 Points)

Walk through your implementation of the indirect lighting function.

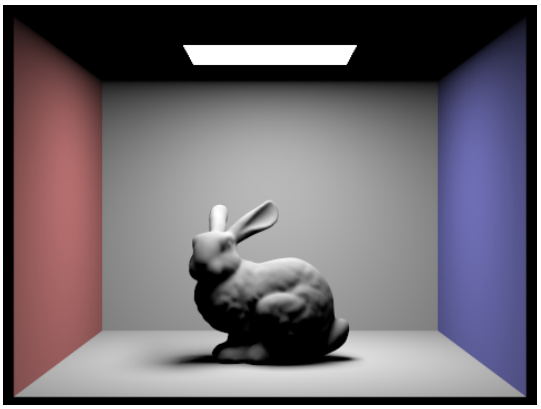
My implementation of the indirect lighting function uses recursion to check light paths that go past just one bounce. I first get the direct-lighting of each given intersection and then based on the probabilty of the russian roulette algorithm, I propagate rays to their next intersection. As long as the object being intersected with isn't a light source or an emitting object, collisions with objects will be intersections. The roulette probabilty is there to handle cases of infinite recursion.

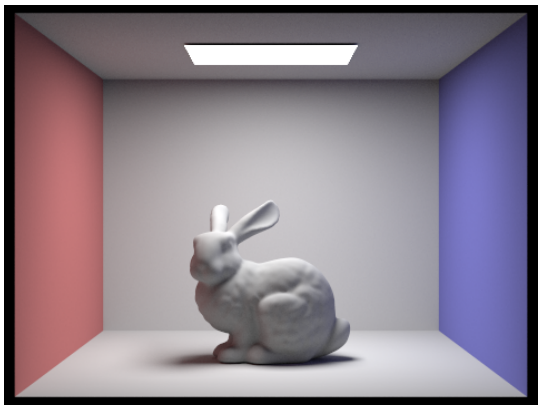
Show some images rendered with global (direct and indirect) illumination. Use 1024 samples per pixel.


CBbunny.dae: 1024 px, 8 light rays, 5 depth


CBspheres_lambertian.dae: 1024 px, 16 light rays, 5 depth

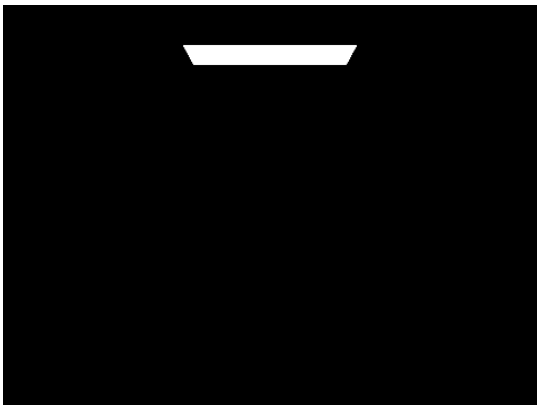
Pick one scene and compare rendered views first with only direct illumination, then only indirect illumination. Use 1024 samples per pixel. (You will have to edit PathTracer::at_least_one_bounce_radiance(...) in your code to generate these views.)

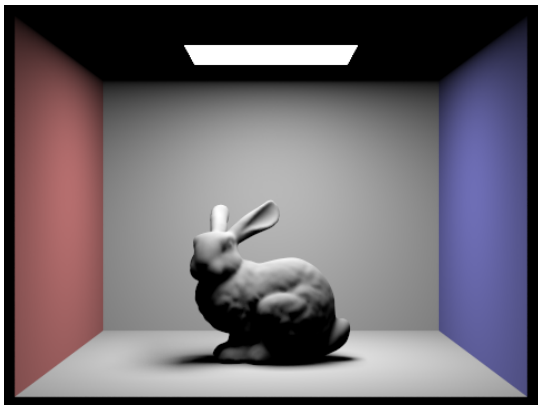

Only direct illumination (CBbunny.dae)

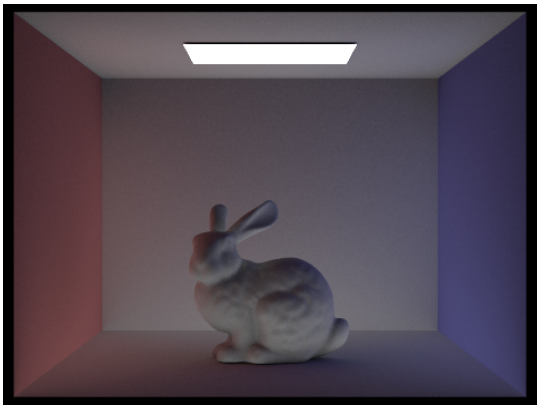

Only indirect illumination (CBbunny.dae)

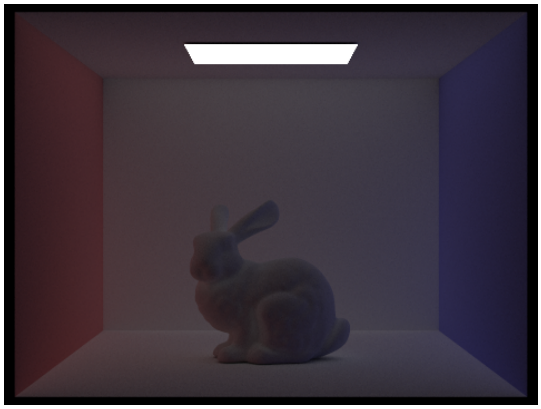
The difference between direct and indirect illumination is notable here because without indirect illumination, the ceiling will not light up. Having an indirect element clearly makes the image more realistic because light beams bounce many times in the physical world.

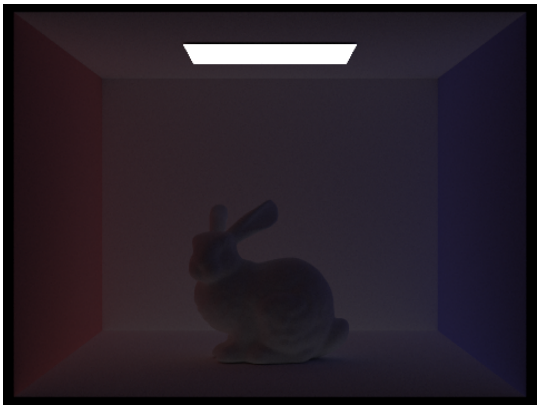
For CBbunny.dae, render the mth bounce of light with max_ray_depth set to 0, 1, 2, 3, 4, and 5 (the -m flag) with isAccumBounces=false. Use 1024 samples per pixel.

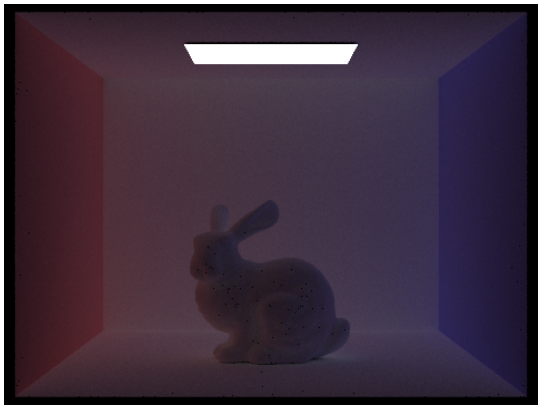

max_ray_depth = 0 (CBbunny.dae)


max_ray_depth = 1 (CBbunny.dae)


max_ray_depth = 2 (CBbunny.dae)

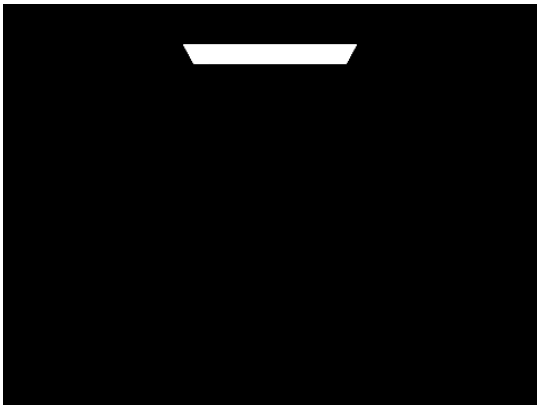

max_ray_depth = 3 (CBbunny.dae)

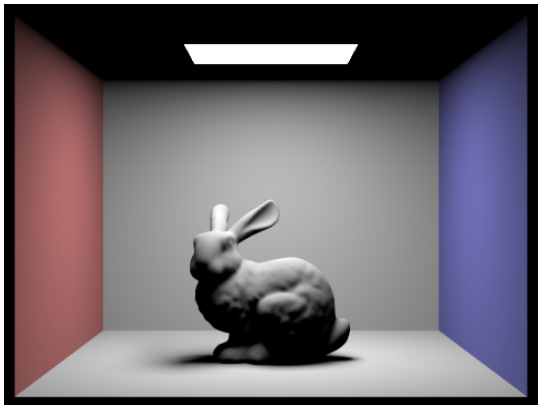

max_ray_depth = 4 (CBbunny.dae)

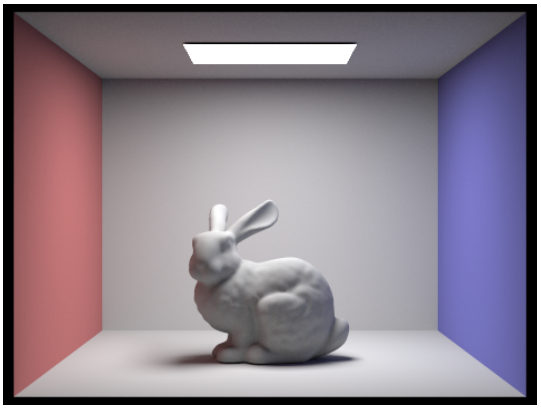

max_ray_depth = 5 (CBbunny.dae)

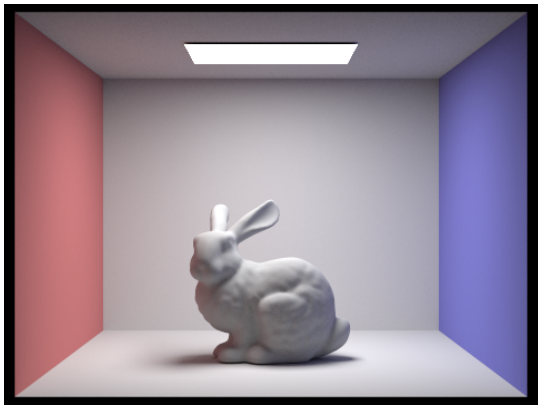
The second and third bounce of light look like dimmer versions of the final result, which is all of these bounces summed together. Unlike the 1-bounce only setting, the two-bounce and three-bounce have the ceiling lit slightly.

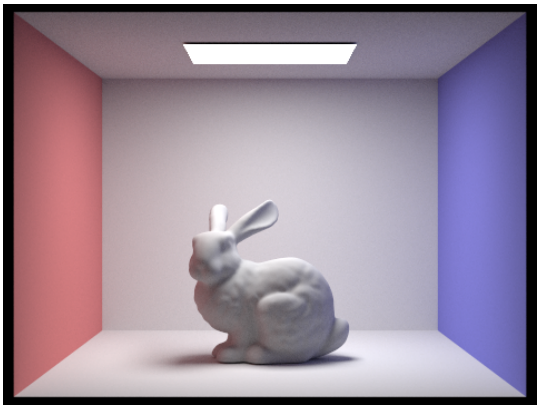
For CBbunny.dae, compare rendered views with max_ray_depth set to 0, 1, 2, 3, 4, and 5 (the -m flag). Use 1024 samples per pixel.

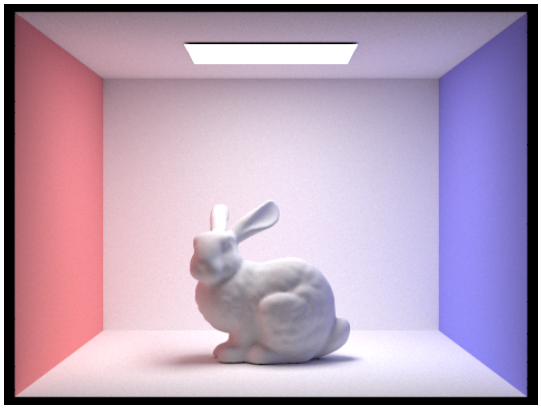

max_ray_depth = 0 (CBbunny.dae)


max_ray_depth = 1 (CBbunny.dae)


max_ray_depth = 2 (CBbunny.dae)

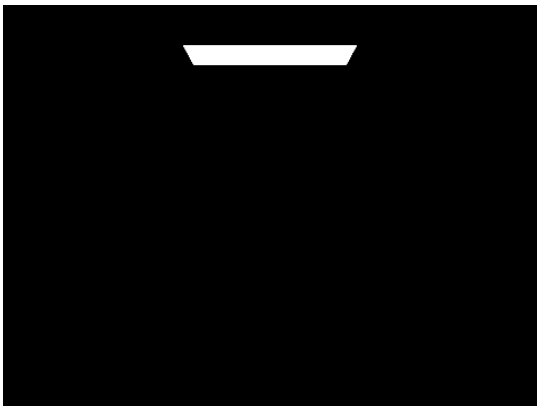

max_ray_depth = 3 (CBbunny.dae)

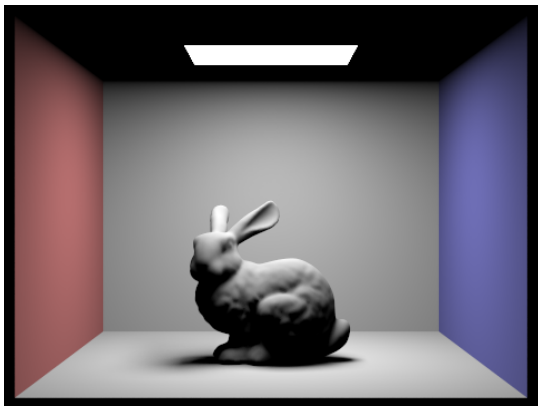

max_ray_depth = 4 (CBbunny.dae)

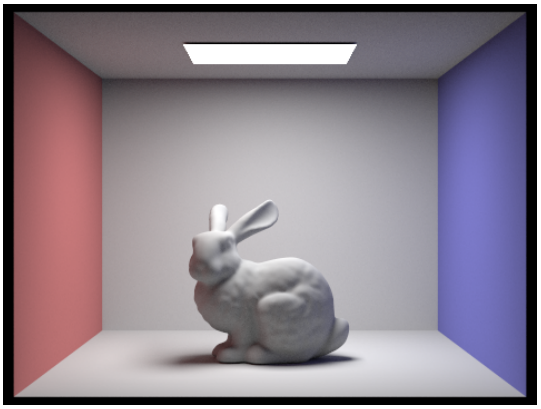

max_ray_depth = 5 (CBbunny.dae)

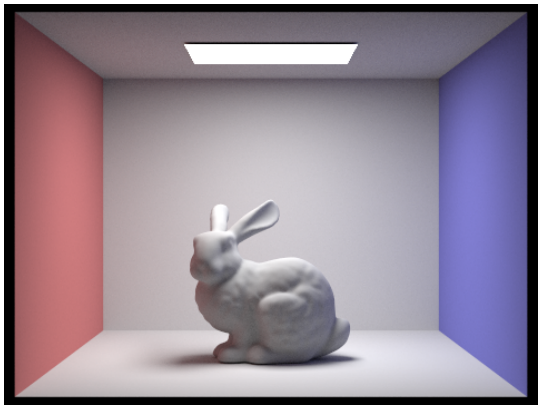
The image gets more lit up as the number of bounces increase.

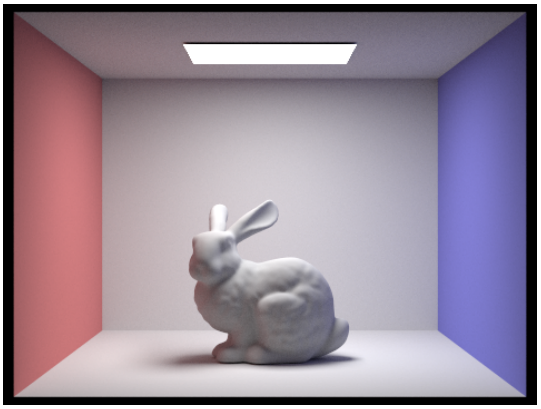
For CBbunny.dae, output the Russian Roulette rendering with max_ray_depth set to 0, 1, 2, 3, and 100 (the -m flag). Use 1024 samples per pixel.

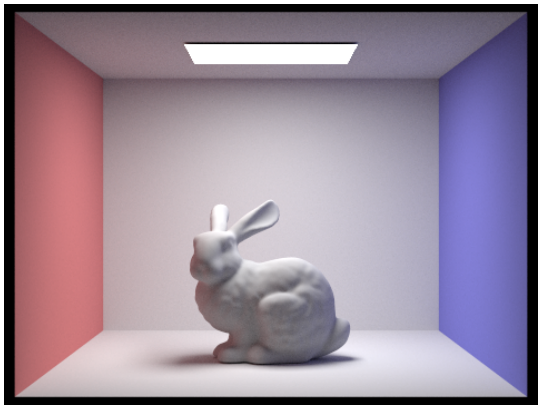

max_ray_depth = 0 (CBbunny.dae)


max_ray_depth = 1 (CBbunny.dae)


max_ray_depth = 2 (CBbunny.dae)

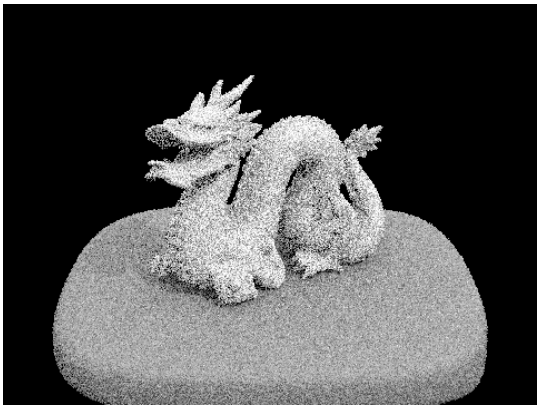

max_ray_depth = 3 (CBbunny.dae)





max_ray_depth = 4 (CBbunny.dae) max_ray_depth = 100 (CBbunny.dae)

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

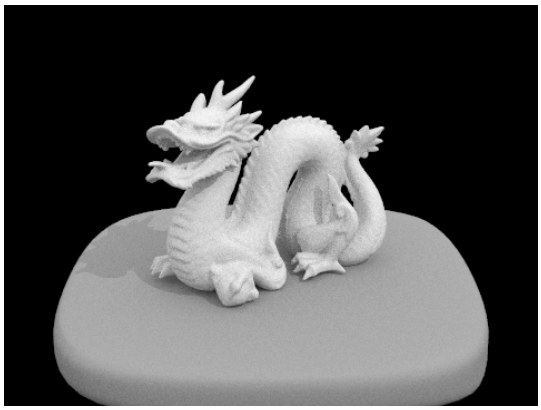

1 sample per pixel (dragon.dae)

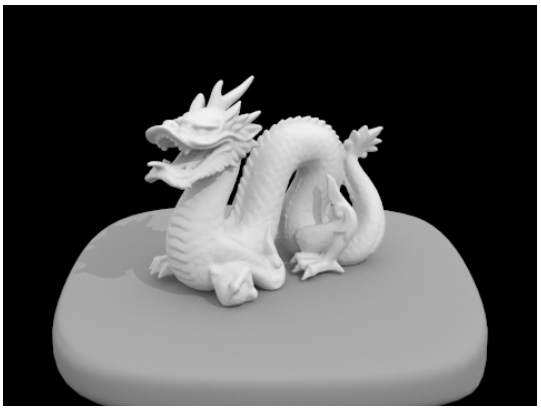

2 samples per pixel (dragon.dae)


4 samples per pixel (dragon.dae)


8 samples per pixel (dragon.dae)


16 samples per pixel (dragon.dae)


64 samples per pixel (dragon.dae)


1024 samples per pixel (dragon.dae)

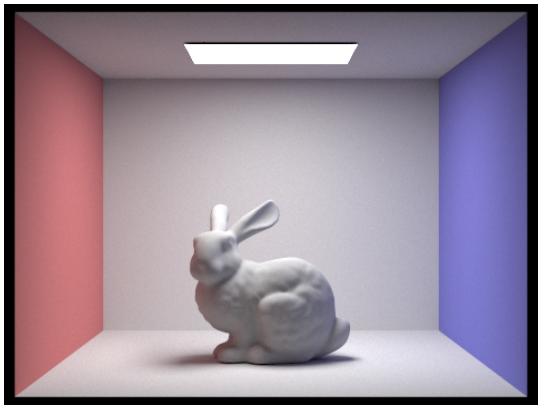
The image gets less noisier as the number of samples per pixel increases.

Part 5: Adaptive Sampling (20 Points)

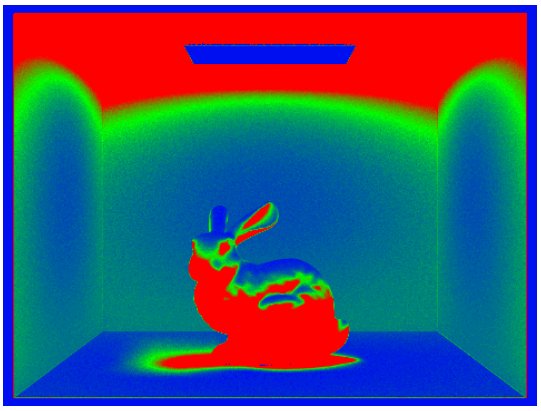
Explain adaptive sampling. Walk through your implementation of the adaptive sampling.

The idea of adaptive sampling is the figure out with pixels converged to values will not change much anymore. These don't require additional sampling and so I can terminate them. This reduces the amount of rays needing to be processed. My adaptive sampling implementations checks if the variance in the samples, the average illuminance, and z-scores from the assumed pdf of illuminance (Gaussian) follow the same pattern as the current rays.

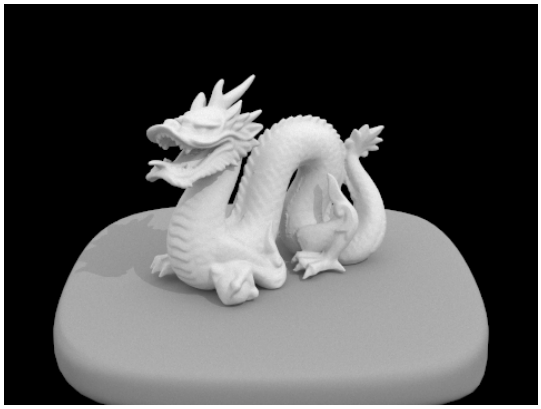
Pick two scenes and render them with at least 2048 samples per pixel. Show a good sampling rate image with clearly visible differences in sampling rate over various regions and pixels. Include both your sample rate image, which shows your how your adaptive sampling changes depending on which part of the image you are rendering, and your noise-free rendered result. Use 1 sample per light and at least 5 for max ray depth.



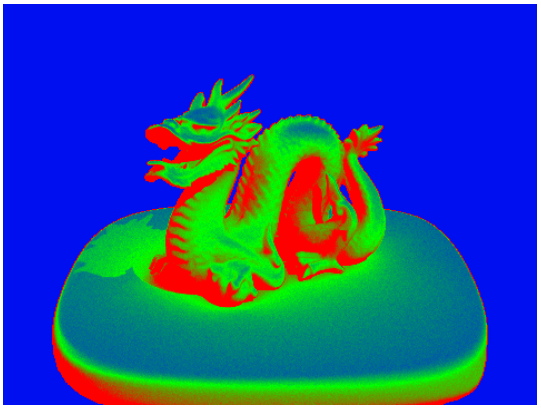
Rendered image (CBbunny.dae), samples per pixel = 2048, max_ray_depth = 5, light samples = 1



Sample rate image (CBbunny.dae)



Rendered image (dragon.dae), samples per pixel = 2048, max_ray_depth = 5, light samples = 1



Sample rate image (dragon.dae)