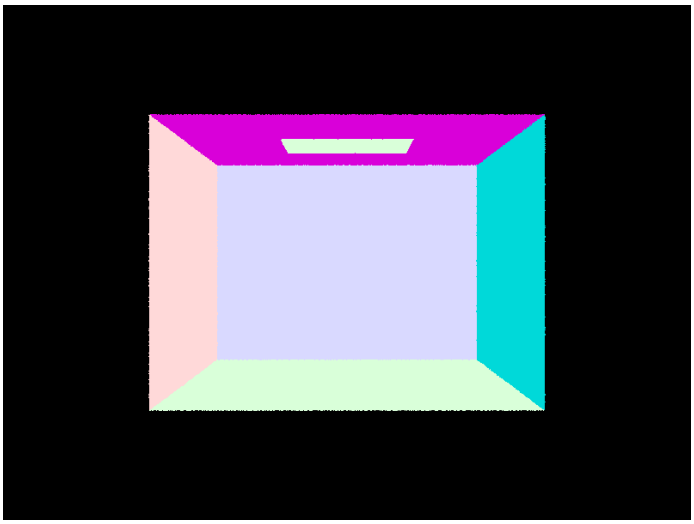
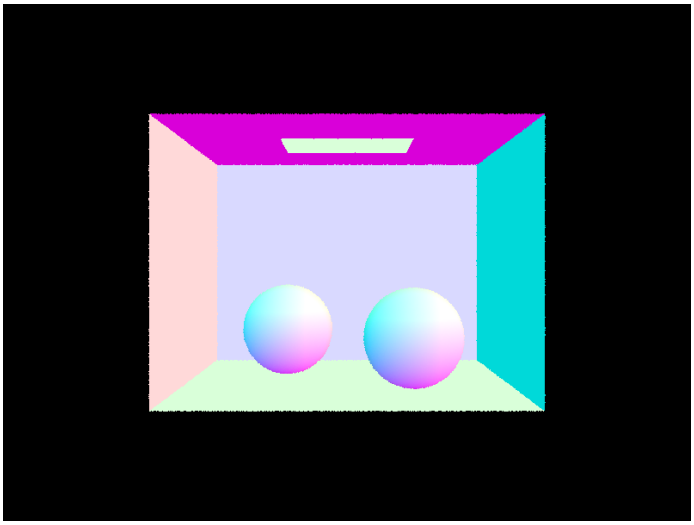
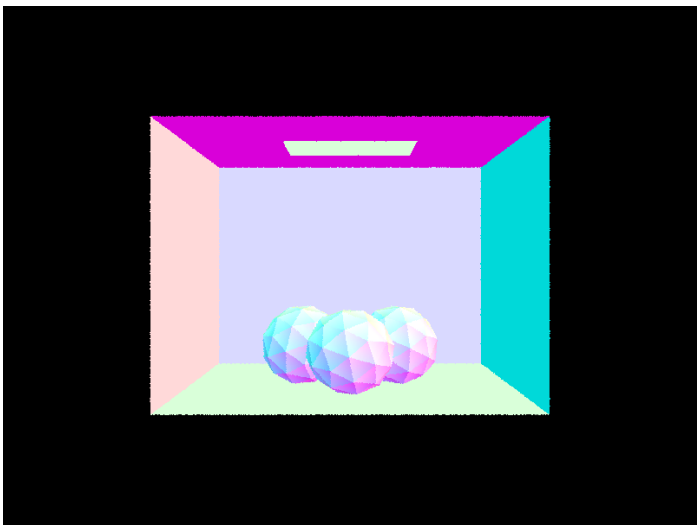
Overview

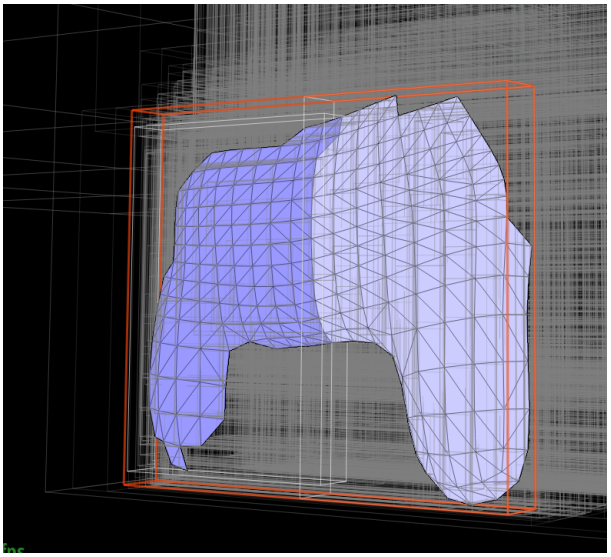
- We started off generating rays from the camera, then implemented algorithms for intersection between rays and shapes. Then we estimated the amount of light hitting a specific point in the scene by the light. We sped this process up by implementing a bounding volume hierarchy. Then we added direct and indirect illumination and adaptive sampling to reduce noise. We encountered many problems like how to handle the light source, too much noise. We solved them by looking at EDstem and reading what other people did to solve the issues.

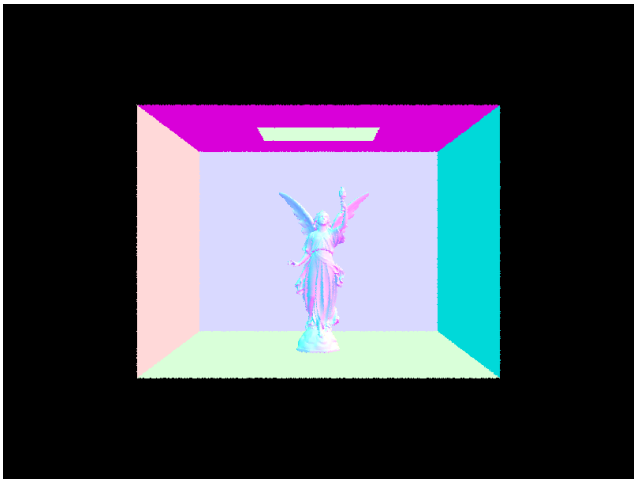
Part 1

- Walk through the ray generation and primitive intersection parts of the rendering pipeline.
    - First, we found the bottom left corner and the top right corner by turning the hFov and the vFov into radians and creating Vector3Ds within the camera space. Next, we found the Camera space's point and used the camera to world rotation matrix to find the direction of the ray. Now we can create the ray using the position of the camera origin as well as a normalized direction that we found. Lastly, we needed to set the min and max t values to n clip and f clip respectively. For the primitive intersections, we had triangle intersections, for which we used the Moller Trumbore algorithm, and sphere intersections, for which we used the quadratic formula to solve for a, b, and c. For the sphere, there would be three scenarios, two solutions, one solution, or no solution for the quadratic formula, which can be represented as intersecting within the sphere, having 2 intersections, intersecting on the surface of the sphere, having 1 intersection, and missing the sphere entirely, having no intersection.
- Explain the triangle intersection algorithm you implemented in your own words.
    - The triangle intersection algorithm using the Moller Trumbore algorithm was mainly creating the vectors from the positions as well as the cross products of the vectors. Then, by plugging in the formula, we were able to obtain t, b1, and b2. t was used to update the intersection, and thus max_t and b1 and b2 were used for barycentric coordinates as coefficients in order to compute the normal of the intersection.
- Show images with normal shading for a few small *.dae* files.

- 
- 
- 

Part 2

- Walk through your BVH construction algorithm. Explain the heuristic you chose for picking the splitting point.
    - First, we would the stopping condition to where the number of primitives we had was less than the max-leaf size. Then we had to split the primitives into the left and the right branches. Our heuristic for picking the splitting point was choosing the average centroid along the longest axis. So the primitives that had a centroid smaller along that axis would be put into the left branch and those larger would be put into the right branch. We partitioned the primitives from start and end and created smaller bounding boxes until it was smaller than the max-leaf size. As for checking the intersection, we would check if it is a leaf, and if it was, we check over the primitives. If it wasn't, we'd recursively call it on the node's branches.
    - 
- Show images with normal shading for a few large *.dae* files that you can only render with BVH acceleration.

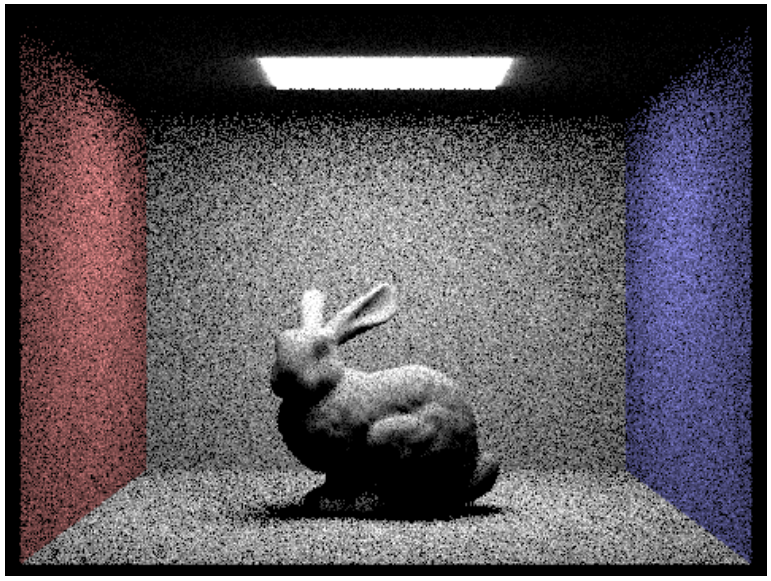
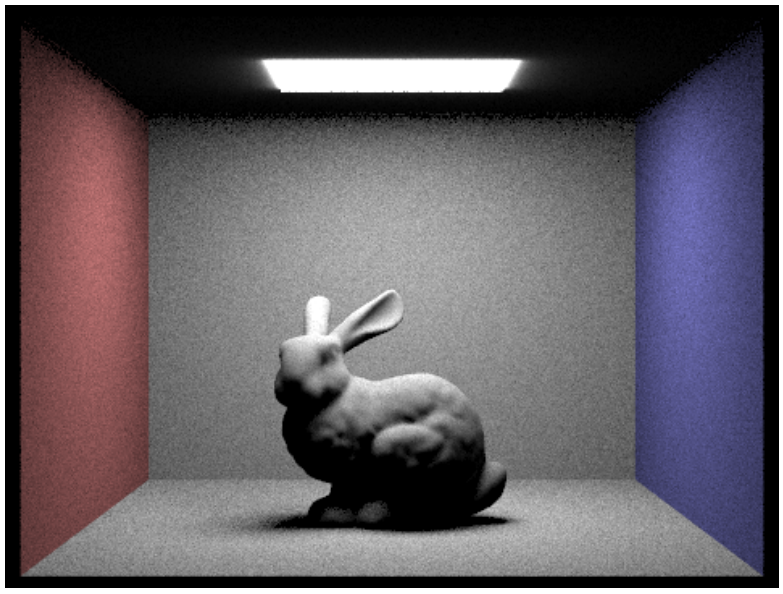
- 
- 
- 

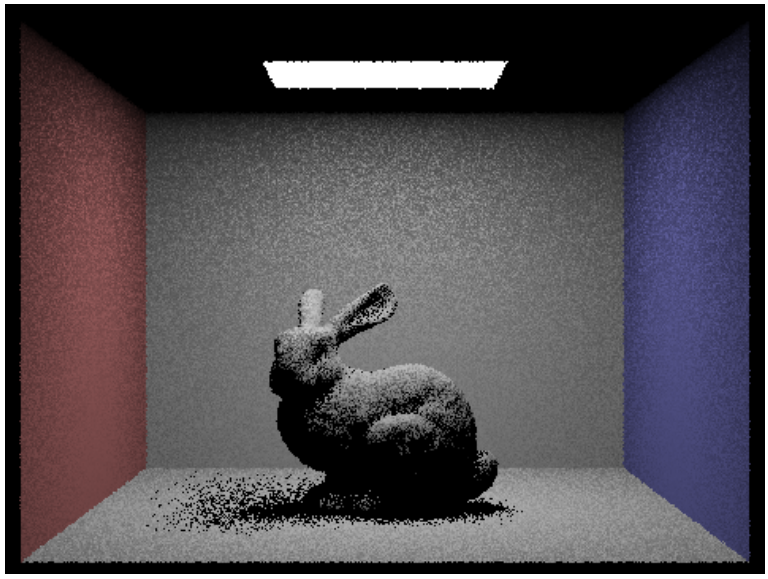
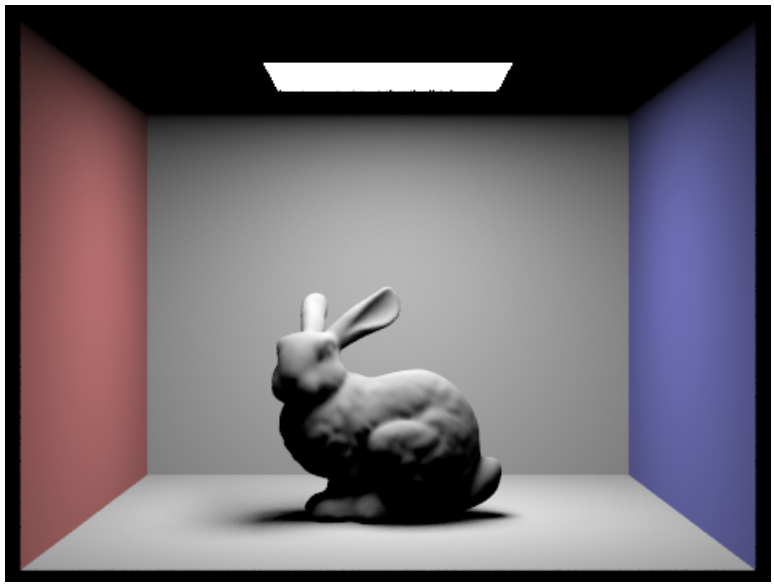
- Compare rendering times on a few scenes with moderately complex geometries with and without BVH acceleration. Present your results in a one-paragraph analysis.
    - The rendering time for the cow was 18 seconds without BVH acceleration and 0.12 seconds with BVH acceleration. The rendering time for Max Planck was 174 seconds without BVH acceleration and 1.1s with BVH

acceleration. The rendering time for CBlucy was over 600 seconds without BVH acceleration and 1.4 seconds with BVH acceleration. We were able to render more complex geometries at a much higher speed than before with BVH. Checking if the rays intersected with the bounding box instead of each individual primitive triangle or sphere saved us time on a large magnitude. This way, we can quickly trim off large branches of primitives within a bounding box if they weren't intersected. This would take our ray complexity from O(n) to O(log(n)).

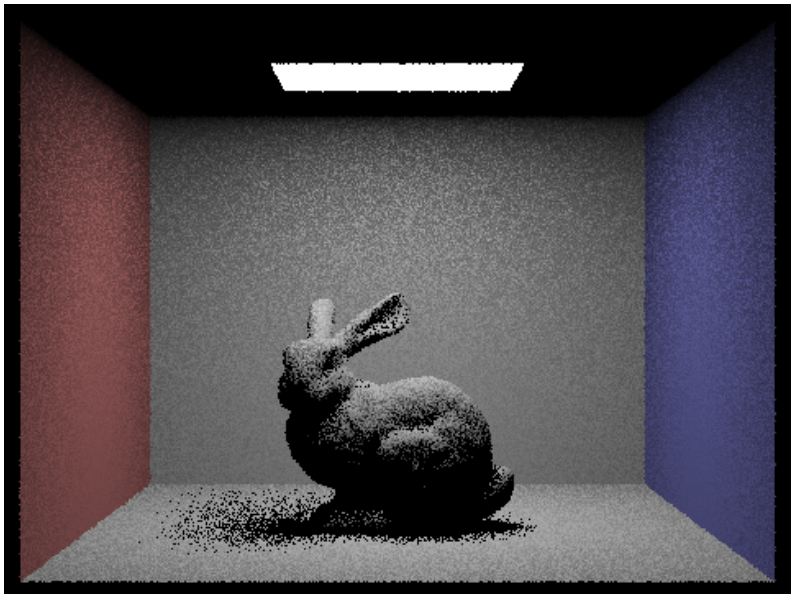
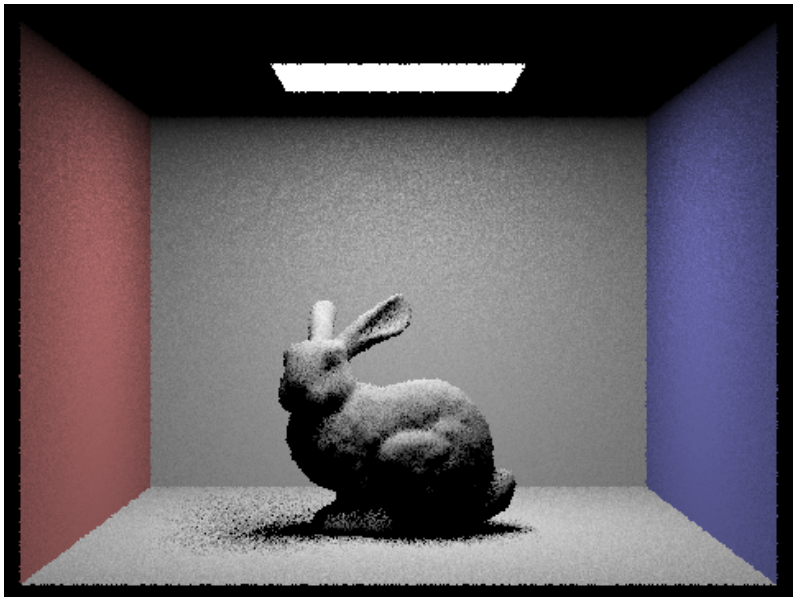
Part 3

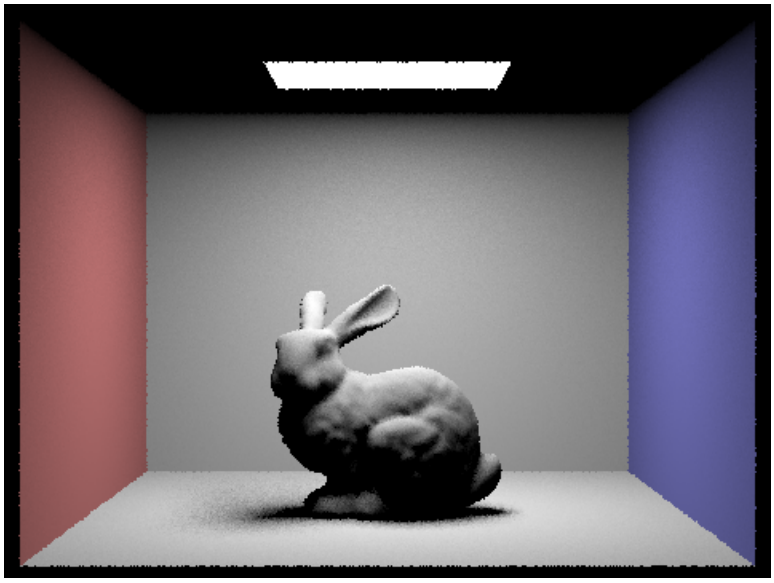
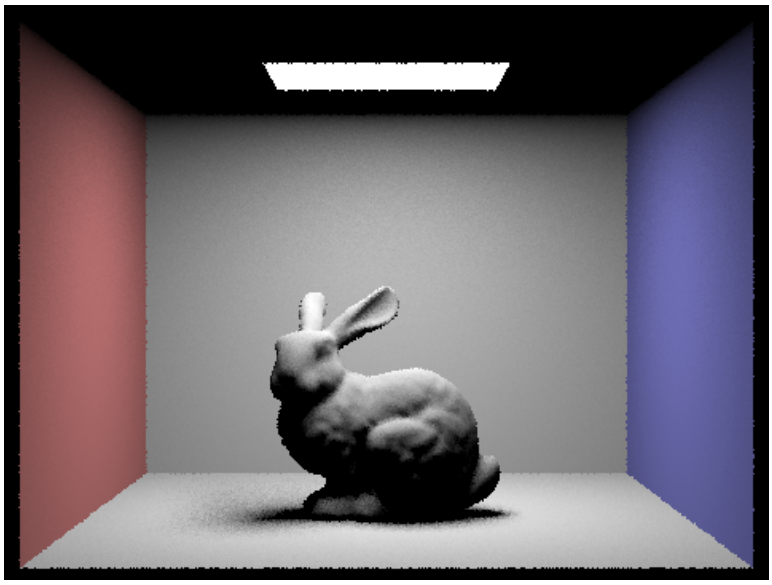
- Walk through both implementations of the direct lighting function.
    - Hemisphere: For each of the number of total samples, we first get the sample from the sampler and convert it into world coordinates. Then we use it to cast a ray from the origin offset by the epsilon constant. After checking if this ray intersects with the bvh, we use the Monte Carlos estimate to find the L out using the zero bounce, bsdf, and cos theta.
    - Importance: We sample all the lights directly for importance. We looped through all the lights in the scene and for each light, we got the sample direction for hit_p and light source and we cast a ray in that direction to check if there is any collision, if not, then we can continue. We will then get the irradiance of the sample through multiplying the radiance of light by BDSF, cosine of the angle of hit_p and ray direction, and dividing by pdf. We summed up the total irradiance and then averaged it out.
- Show some images rendered with both implementations of the direct lighting function.
    - Hemisphere

- 
- 
  - Importance

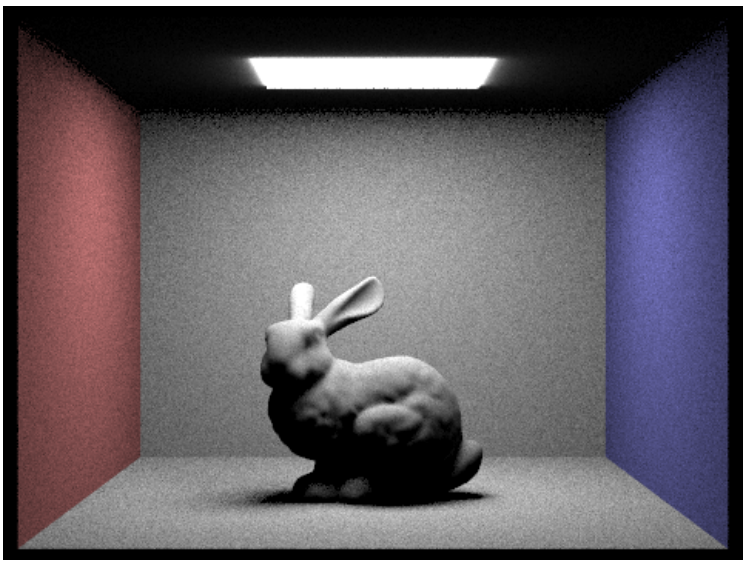
- 
- 

- 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.

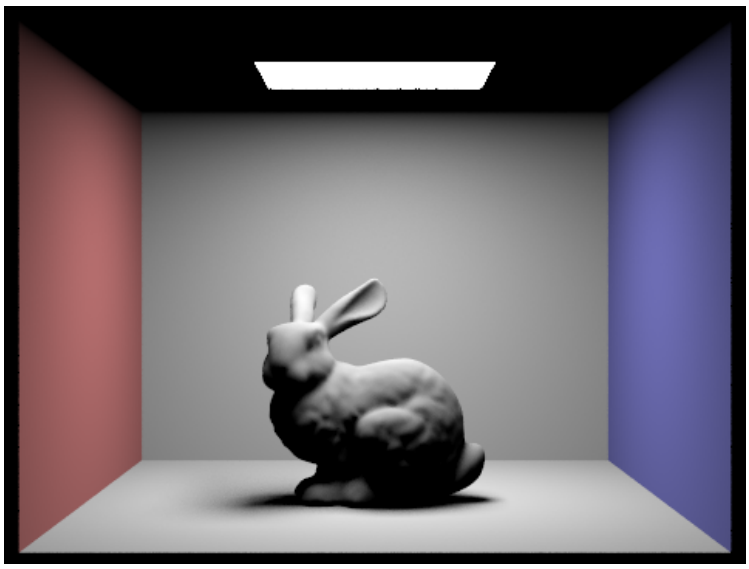
- 1: 
- 4: 

- 16: 
- 64: 

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



Hemisphere Sampling

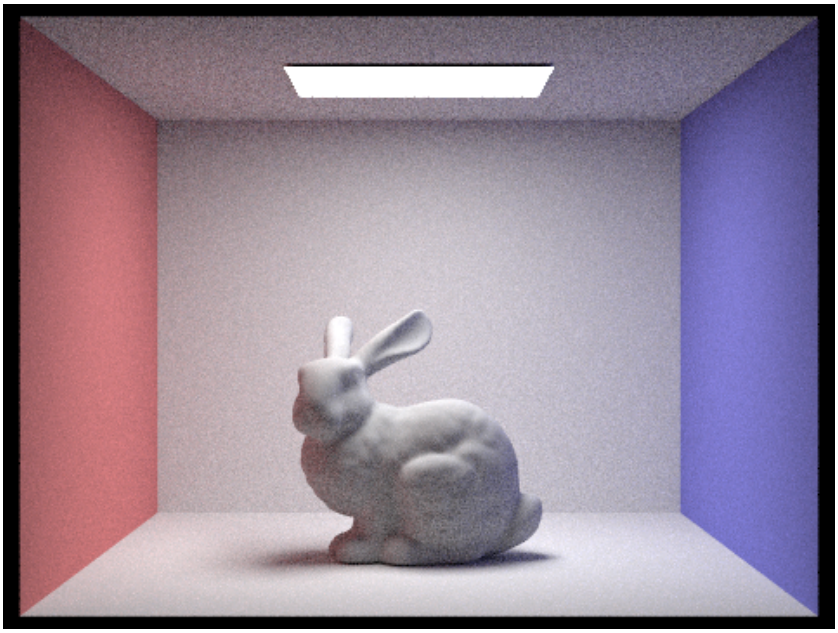
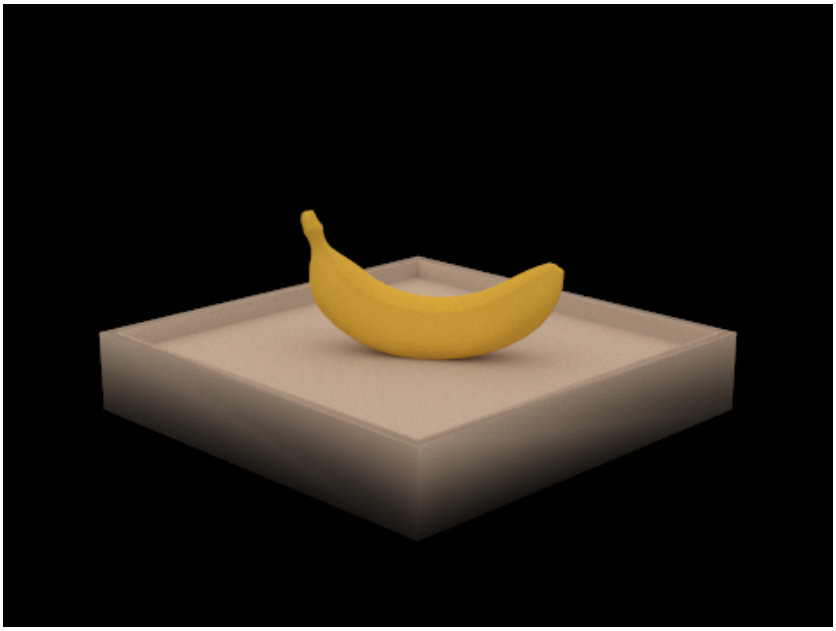


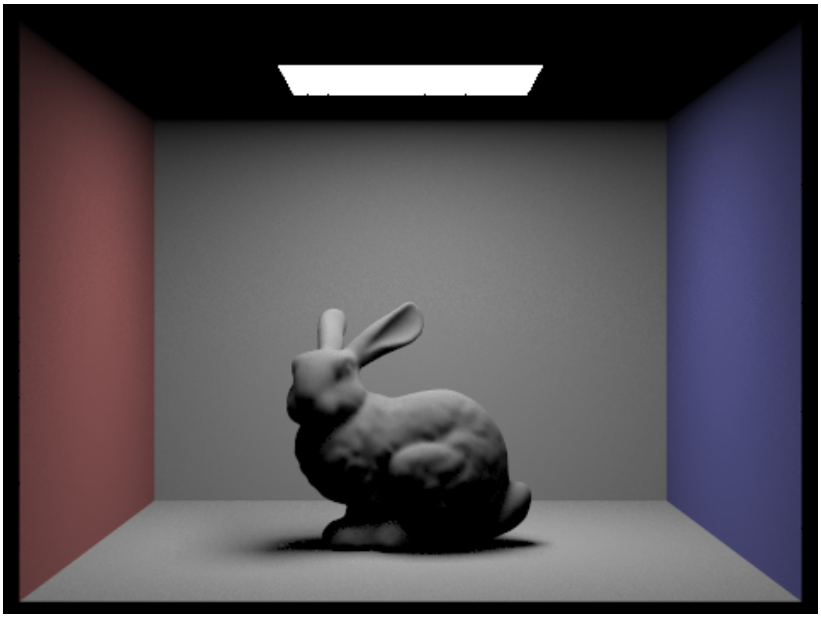
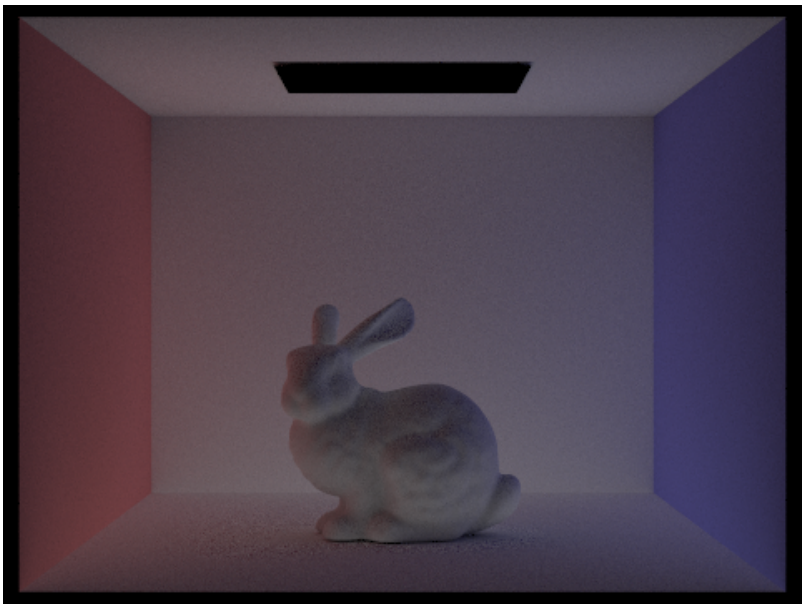
Importance Sampling

- These two sampling methods have some differences between the two images. The Uniform Hemisphere sampling is more grainy and has an effect around the light while the importance sampling looks smoother with no effect around the light. This is because with uniform hemisphere sampling, we are taking samples in all directions around an origin, meaning that some of those don't have much light, so there are some dark points. For importance sampling, we prioritize samples where there could be light, and thus there are fewer dark points, creating a smoother image.

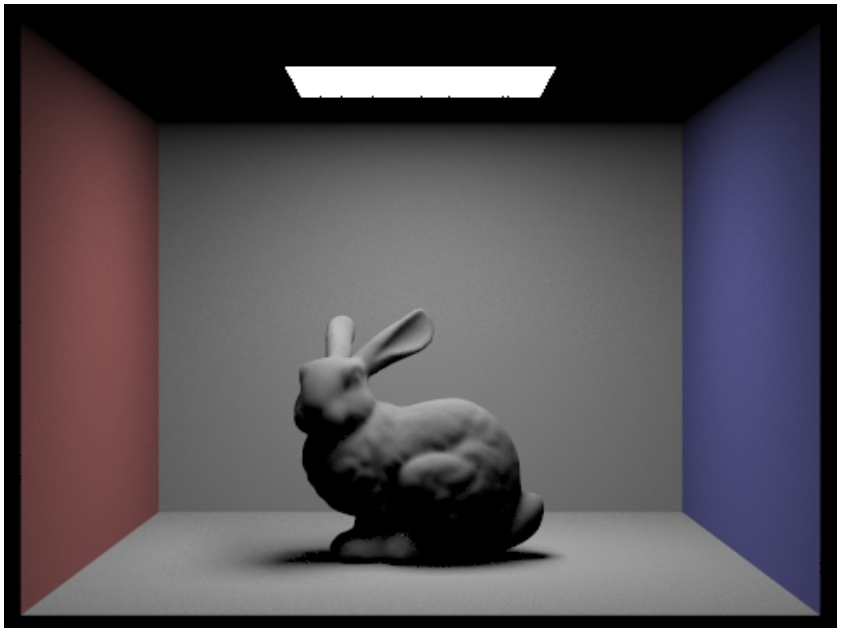
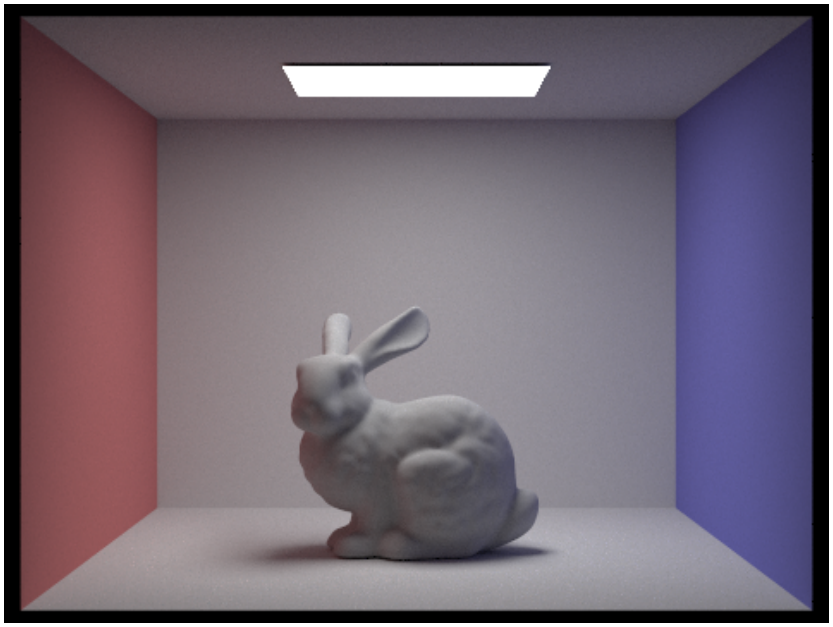
Part 4

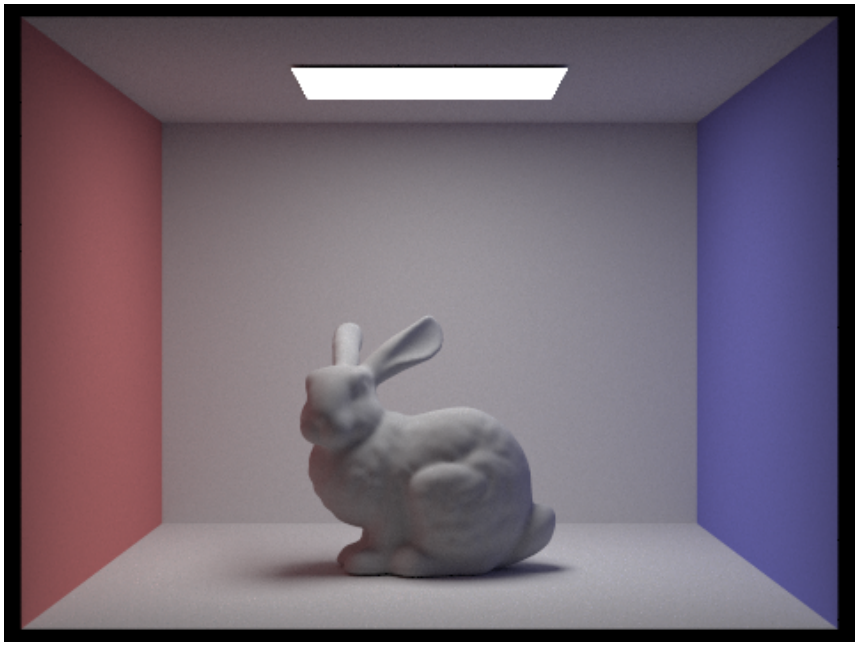
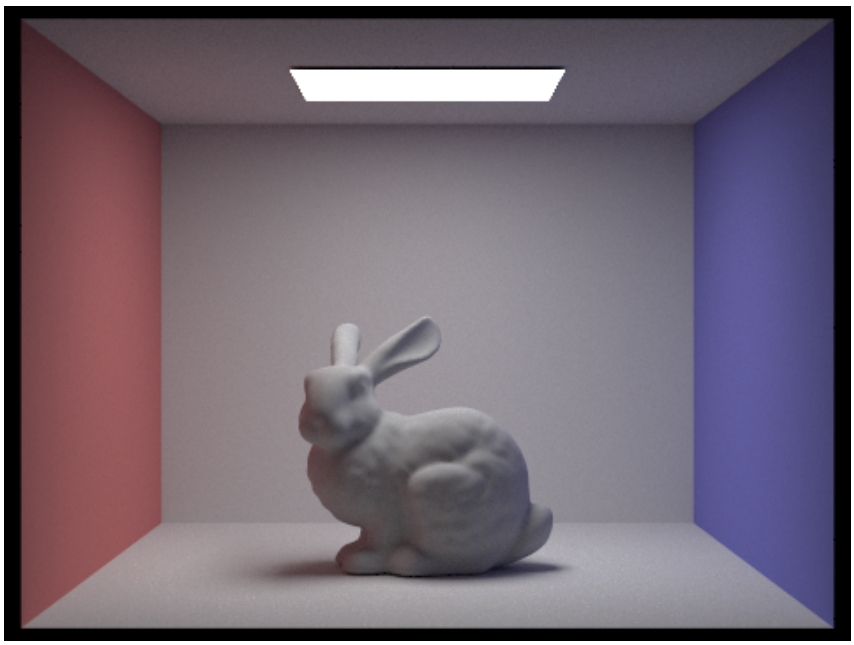
- Walk through your implementation of the indirect lighting function.
  - Indirect lighting accounts for more than one bounce of light. We had to add both the radiance at zero bounces as well as at least one bounce. The biggest challenge was to implement the Russian Roulette to randomly determine which light rays will terminate and which will not. We start off with the one bounce radiance, then append the additional bounces using Russian Roulette with a terminating probability of 0.35 as suggested. With this, we calculated the ray from hit_p given and check for intersection while recursively calling at_least_one_bounce_radiance with a modified depth value of one less to keep track of the bounces. Using BSDF, we can calculate the irradiance from indirect light through using the dot product of wi and the normal vector, pdf, and cpdf.

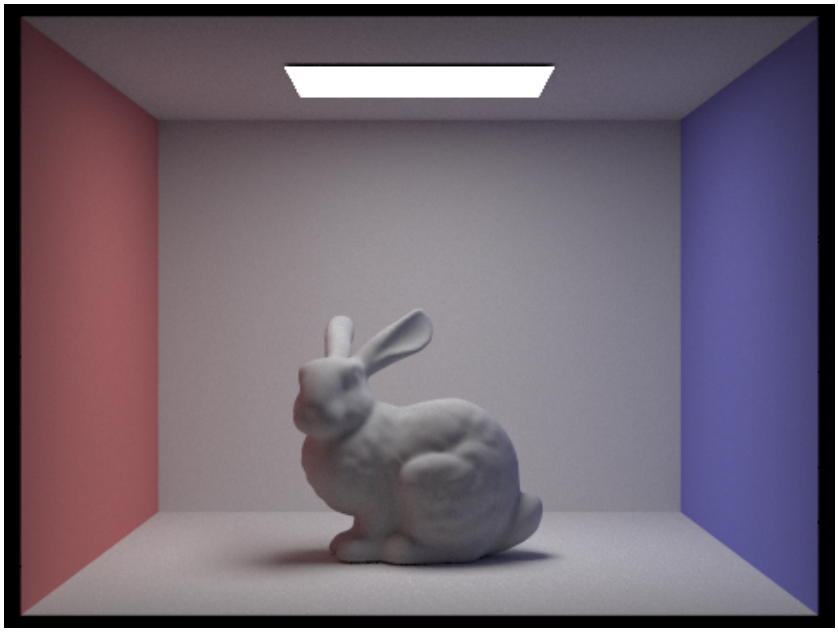
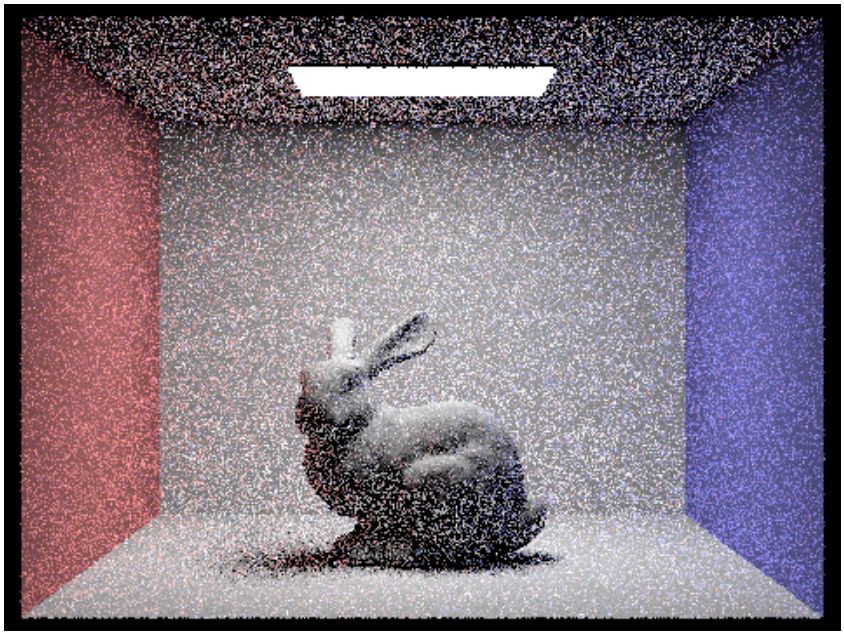
- Show some images rendered with global (direct and indirect) illumination. Use 1024 samples per pixel.
  - 
  - 
  -
- Pick one scene and compare rendered views first with **only** direct illumination, then **only** indirect illumination. Use 1024 samples per pixel. (You will have to edit `PathTracer::at_least_one_bounce_radiance(...)`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

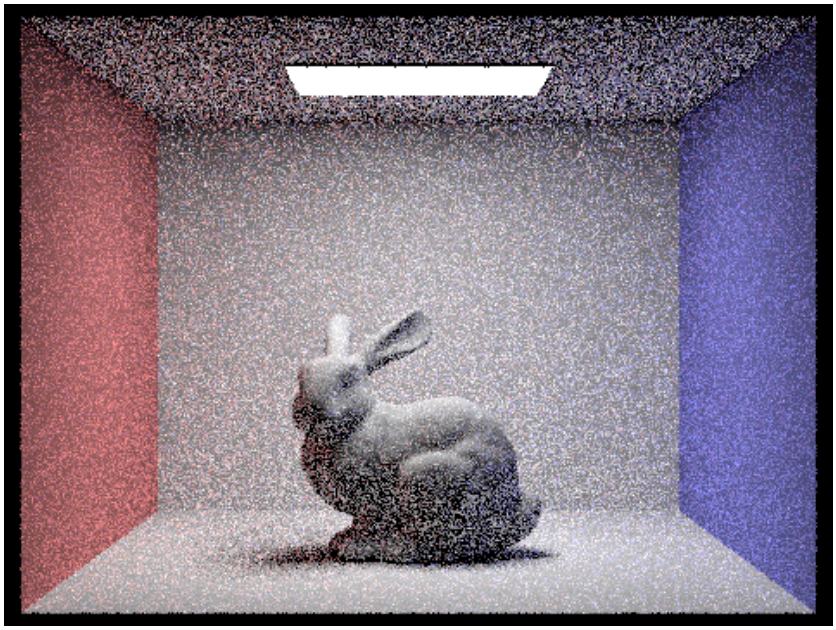
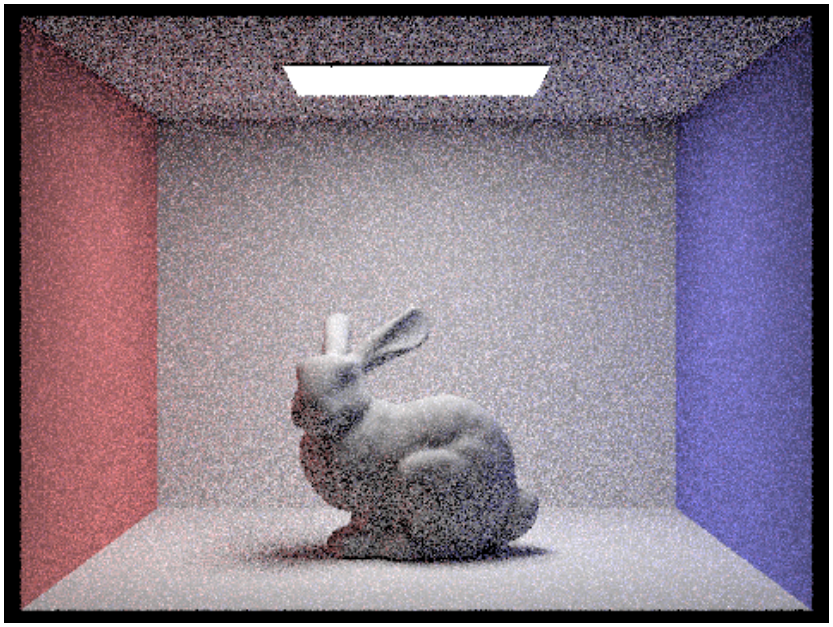
- Direct: 
- Indirect: 

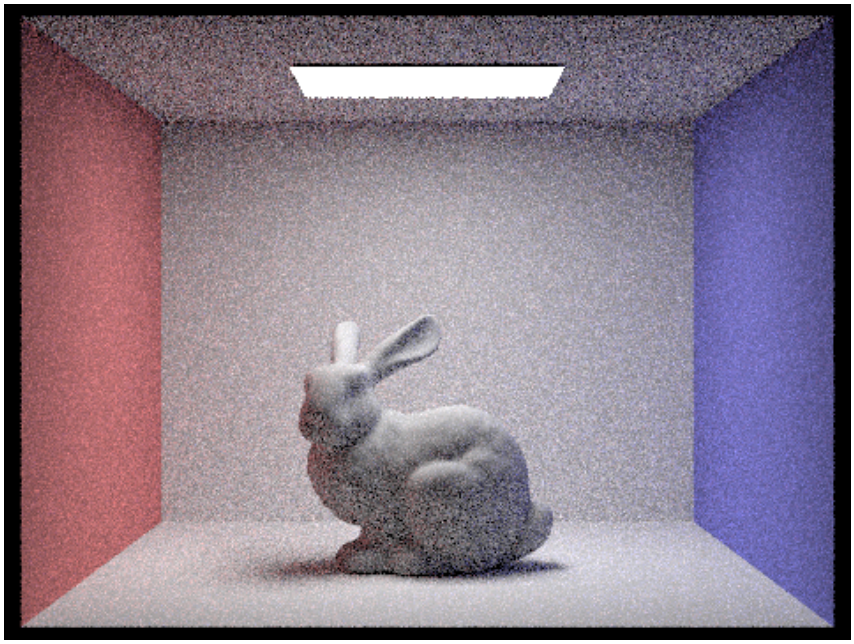
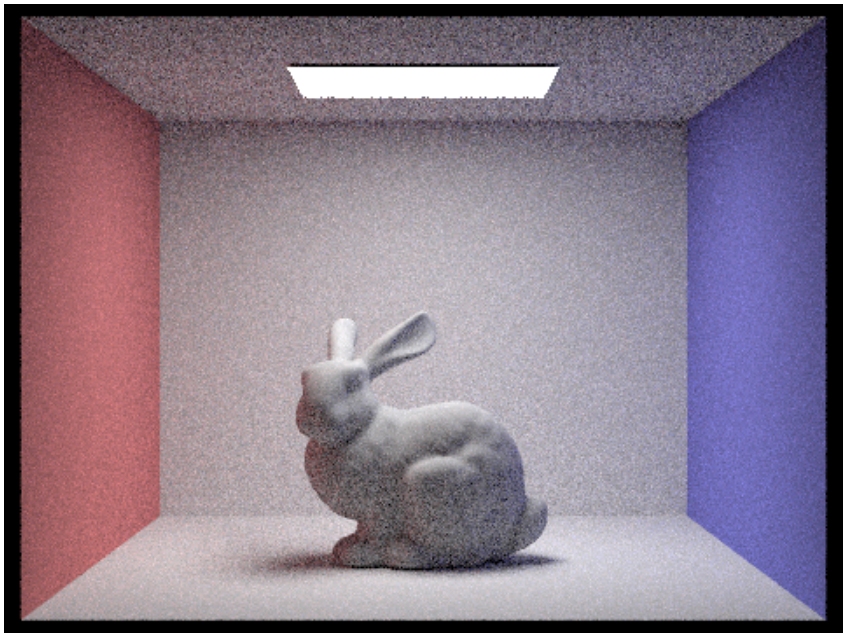
- For *CBbunny.dae*, compare rendered views with `max_ray_depth` set to 0, 1, 2, 3, and 100 (the `-m` flag). Use 1024 samples per pixel.

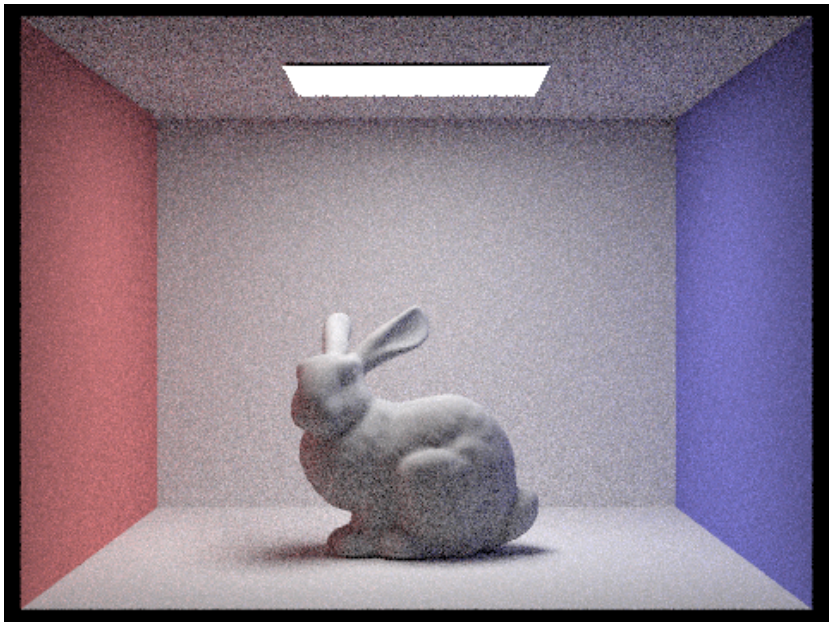
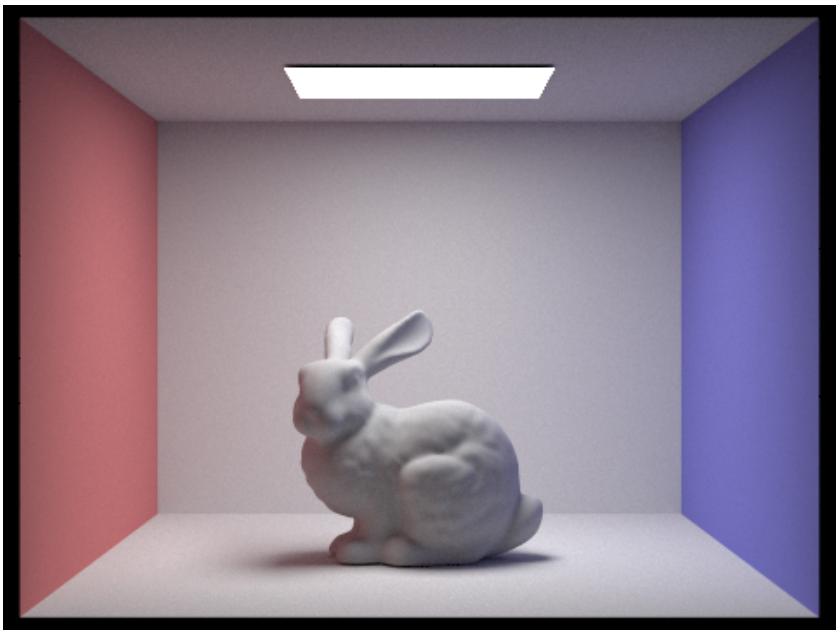
- 0: 
- 1: 

- 2: 
- 3: 

- 100: 
- 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: 

- 2: 
- 4: 

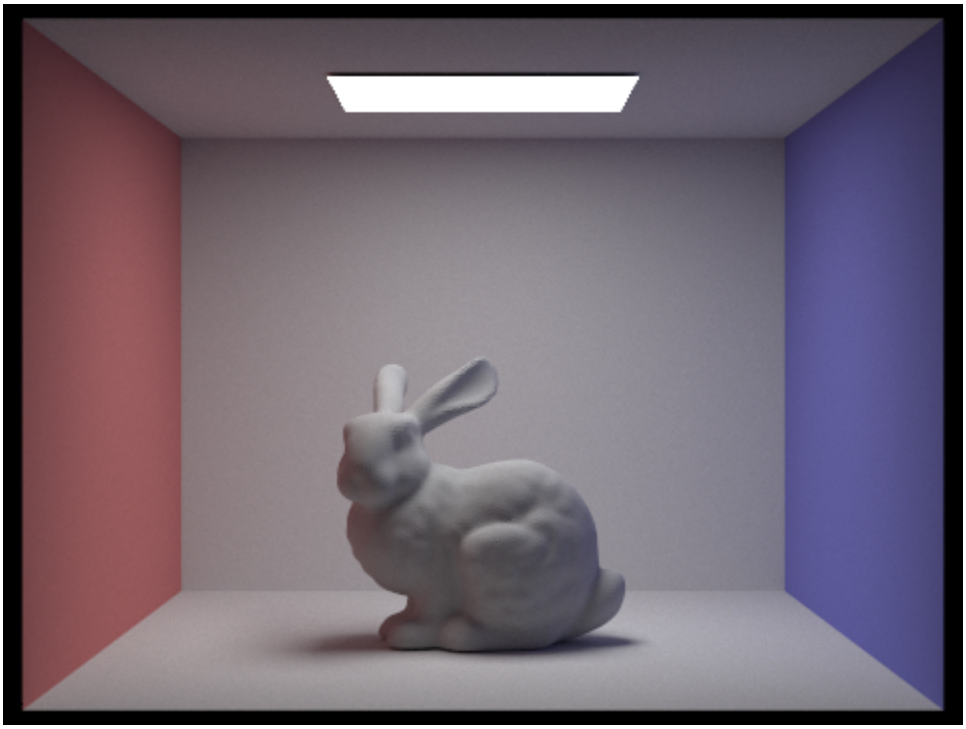
- 8: 
- 16: 

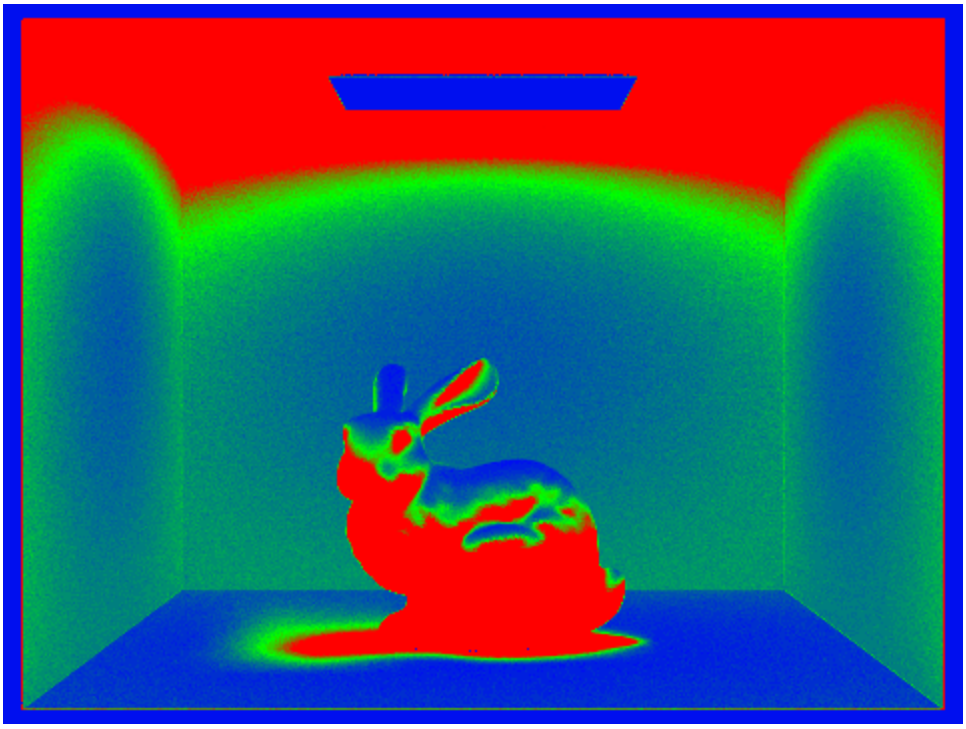
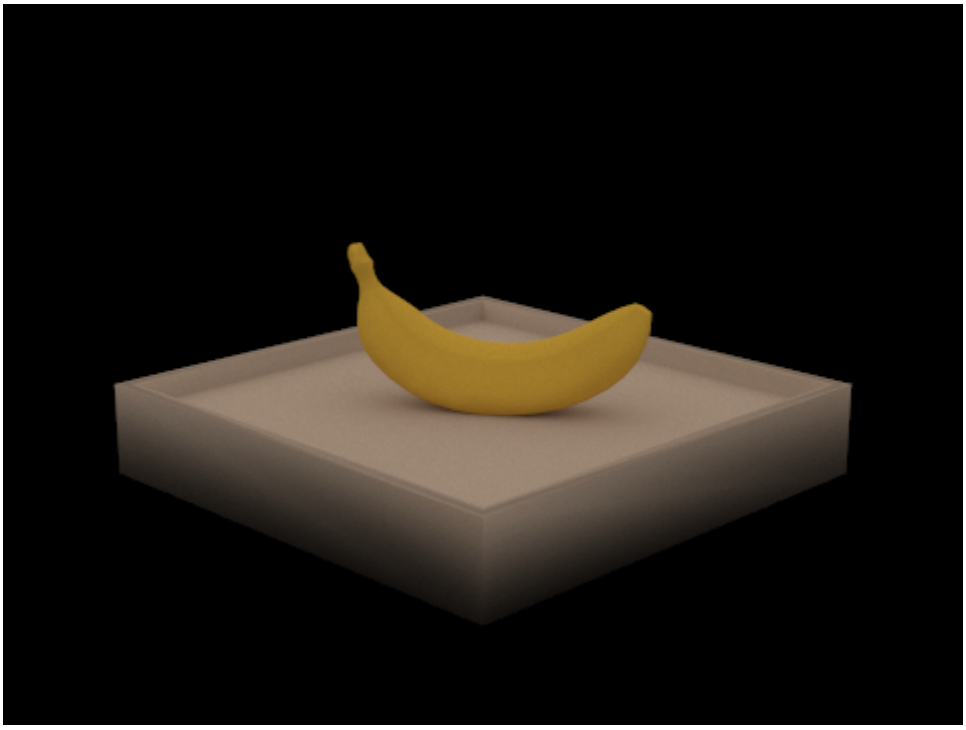
- 64: 
- 1024: 

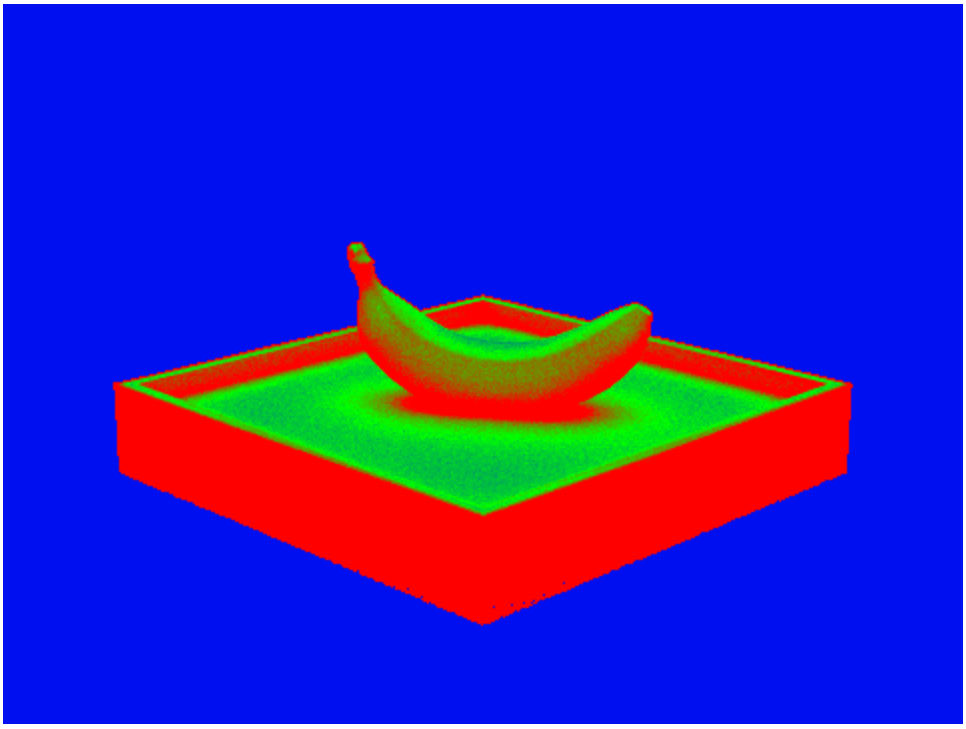
Part 5

- Explain adaptive sampling. Walk through your implementation of the adaptive sampling.
    - Adaptive sampling tries to avoid the process of using a fixed (high) number of samples per pixel by concentrating the samples in the part of the image that needs it (have the most noise). Essentially, we increase the number of samples for only a certain subset of the total pixels. In each pixel, the illuminance values should be normally distributed. Thus, we can

use z-test – 1.96 * (std / sqrt(sample #)) – to determine with a 95% accuracy whether a pixel has converged (if I <= max tolerance * mean).

- 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.
  - 

- 
- 

- 

At the end, if you worked with a partner, please write a short paragraph together for your final report that describes how you collaborated, how it went, and what you learned.

- We did number 1, 3, and 4 first because 2 looked really daunting. But then we realized that if we don't finish 2, it will take too long to render. So we spent most of our effort doing 2. Matt spent the most time debugging number 2 and number 3. David spent the most time debugging number 3 and 5. We should have started earlier because rendering took way too long for number 5. There were a few bugs like the lights not working and the entire bunny being red, but we were able to easily debug and fix it. We will start the project earlier next time.