CS 184: Computer Graphics and Imaging, Spring 2024

Project 3-1: Path Tracer

Anav Mehta

Website URL: Website

Overview

In this Project, I implemented a ray tracing pipeline, starting with generating sample camera rays for each pixel in the image. To determine whether a ray intersected a primitive, I incorporated different ray-object intersections such as ray-triangle and ray-sphere. To improev on the rendering speeds, we constructed a Bounding Volume Heirachy where instead of checking if a ray intersects each primitive we check against a larger, encompassing bounding volume. Initially the rendering only used normal shading; we expanded on this by implementing two direct illumination techniques one being sampling from a uniform hemisphere and another being importance sampling of light sources. These methods introduced zero and one bounce illumination creating a more realistic scene. We then built on the illumination techniques by implementing global illumination, allowing light to reflect more than once in a scene. The potential bias introduced limiting the number of bounces was combatted through Russian Roulette global illumination, which probabilistically terminated light bounces with a 30% likelihood. Lastly, adaptive sampling was adopted to reduce noise in the rendering, prioritizing areas requiring more intensive sampling to achieve convergence. I faced plenty of debugging problems in this project the main ones being not allocating the vectors on the heap when constructing the BVH, forgetting to normalize when notaccumulating bounces, and dealing with object and world coordinate spaces.

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

Walk through of the ray generation and primitive intersection.

To generate one ray, we first had to map the image space coordinates onto the sensor in camera space. Since (0, 0) and (1, 1) represent the sensors bottom left corner `(-tan(0.5 * hFov_radians), -tan(0.5 * vFov_radians))` and top right corner `(tan(0.5 * hFov_radians), tan(0.5 * vFov_radians))` respectively, I was able to properly map through two formulas. The camera_space_x coordinate = `(sensor_top_x - sensor_bottom_x) * x + sensor_bottom_x` and camera_space_y = `(sensor_top_y - sensor_bottom_y) * y + sensor_bottom_y`. (camera_space_x, camera_space_y, -1) represented the direction of the generated ray. We normalize the direction and convert to world space. We finally generate the ray and set the min_t and max_t of the ray to the near clip and forward clip respectively. To raytrace a pixel, we then generate ns_aa rays to estimate the radiance for that pixel.

Triangle intersection algorithm.

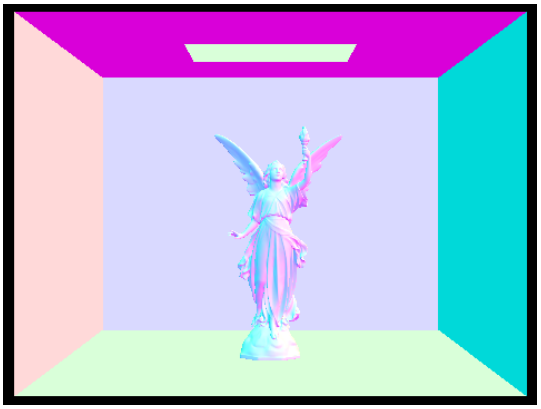
To determine if a ray intersected a triangle, I implemented the Möller-Trumbore Algortihm. By solving the matrix shown below we were able to solve for the time the ray intersected the triangle and barycentric coordinates b1 and b2. If the time intersected was between the bounds of ray min and max time and b1, b2, and (1 - b1 - b2) > 0 and less than 1 then we can confirm it is an intersection and update the max_t of the ray. With our barycentric coordinates we can interpolate the surface normal using `(1 - b1 - b2) * n1 + b1 * n2 + b2 * n3`.

$$\begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = \frac{1}{\vec{S}_1 \bullet \vec{E}_1} \begin{bmatrix} \vec{S}_2 \bullet \vec{E}_2 \\ \vec{S}_1 \bullet \vec{S} \\ \vec{S}_2 \bullet \vec{D} \end{bmatrix}$$

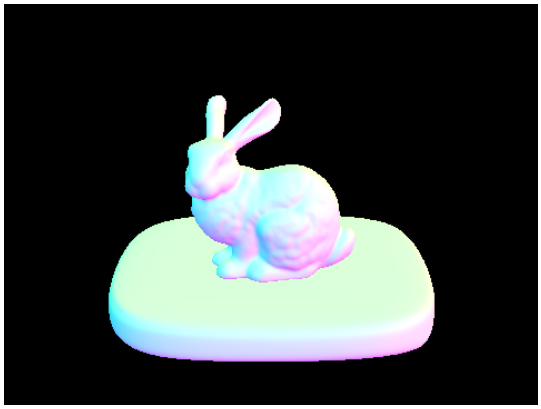
Sphere intersection algorithm.

To determine if a ray intersected a sphere, we can check for times where the (o + td - c) - R^2 = 0 where o + td is the ray formula, c is the origin of the sphere, and R is the radius of the sphere. Since we know this will result in the form at^2 + bt + c, we can solve for t by finding the coefficients a, b, c. Expanding out the terms, we get a = dot(d, d), b = 2*dot(o - c, d), and c = dot(o - c, o - c) - R^2. We use the quadratic formulas to find the times where the ray enters and exits the sphere denoted t1, t2 respectively. If the t1 and t2 are both valid, we can set the rays max t to be the smaller of the two. We can then find the surface normal of the intersection using the formula (p - o).unit().

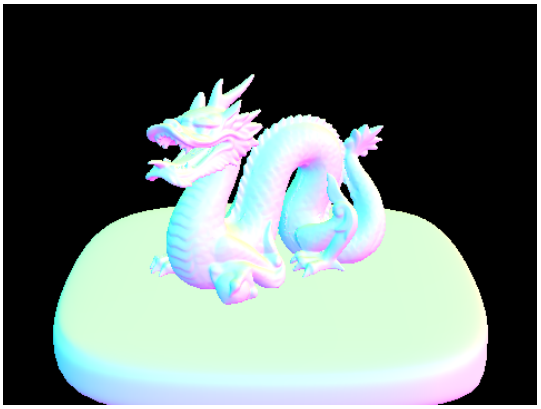
Show images with normal shading for a few small .dae files.



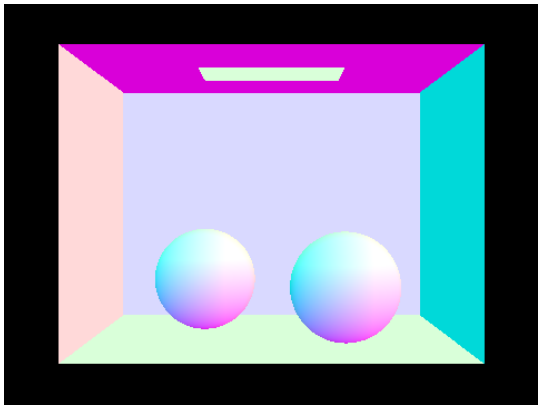
Normal Shading of Lucy



Normal Shading of a Bunny



Normal Shading of a Dragon



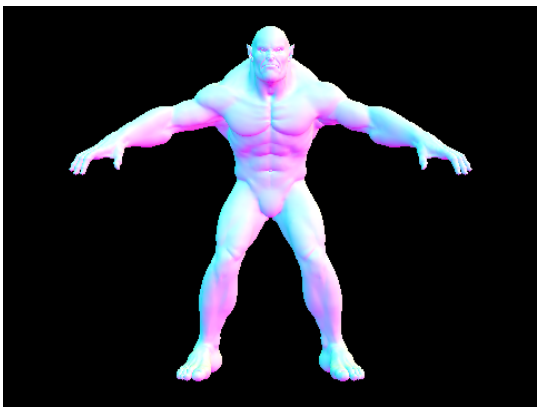
Normal Shading of Spheres

Part 2: Bounding Volume Hierarchy (20 Points)

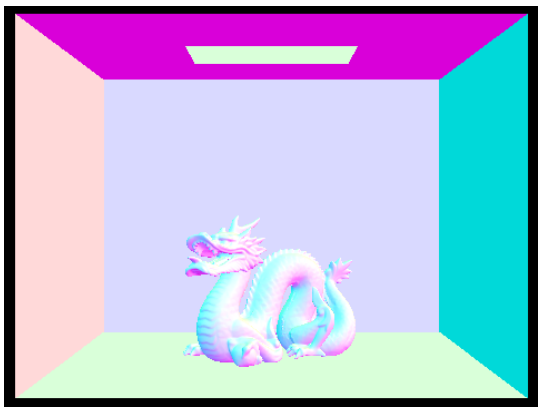
Walk through your BVH construction algorithm. Explain the heuristic you chose for picking the splitting point.

To construct the BVH, we initiate a recursive process and for each recursive call we create a bounding box and expand it for each primitves bounding box while simultaneously, we summming each primitives bounding boxes x, y and z. A new BVHnode is then created. For a leaf node—identified when the primitive count is less than the max_leaf_size-we just set the nodes start and end pointer to the passed in params. For inner nodes, we have to partition the primitives into leftSet or rightSet based on each axis. The heuristic I chose to split on was the difference between the primitives stored in the leftSet and rightSet splitting on the axis that had the most balance between the two sets. We then recursively call the construction algorithm on the chosen left and right sets for the nodes left and right child.

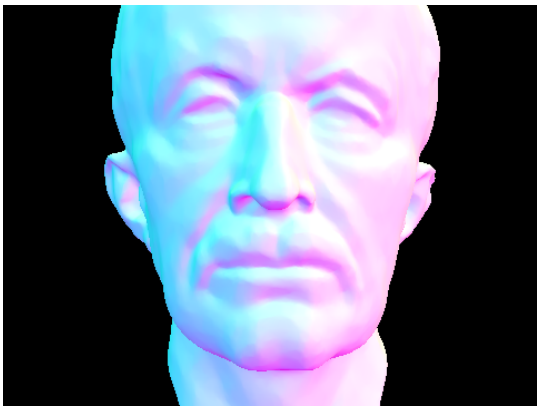
Show images with normal shading for a few large .dae files that you can only render with BVH acceleration.



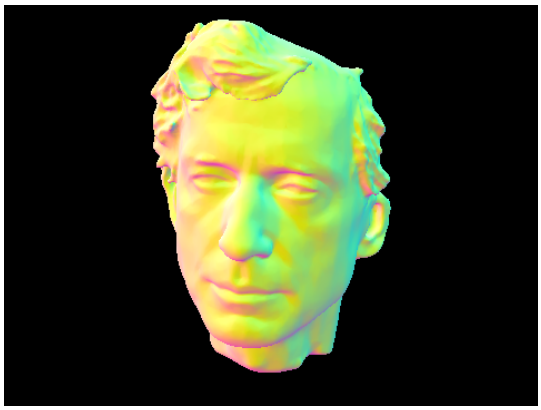
BVH acceleration rendering of Beast



BVH acceleration rendering a Dragon



BVH acceleration rendering of Max Planck



BVH acceleration rendering of Peter

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

To show the acceleration that BVH provides, I rendered three scenes with and without BVH acceleration all rendered on a M1 Mac of resolution 480x360. The first was cow.dae where without BVH acceleration took 17.4257s while with BVH acceleration it took 0.0466s. The second was beetle.dae where without BVH acceleration took 21.7633s while with BVH acceleration it took 0.0961s. And finally building.dae where without BVH acceleration took 121.8985s while with BVH acceleration it took 0.0445s.

Part 3: Direct Illumination (20 Points)

Walk through both implementations of the direct lighting function.

The two direct lighting functions we implemented were direct lighting through uniform hemisphere sampling and direct lighting through importance sampling.

Uniform Hemisphere Sampling

In uniform hemisphere sampling, an incoming ray direction is sampled across the hemisphere. This involes selecting a direction wj and transforming it to world space. The BRDF is then calculated using the incident light direction wj and the outgoing light direction w_out. This uniform sampling means that each direction has an equal probability of being chosen, hence our probability density function for this sampling strategy is 1/2π. The irradiance contribution from a light source in the direction w_j is proportional to the cosine of the angle between the surface normal N and w_j which we can calculate the dot product between these two vectors. We

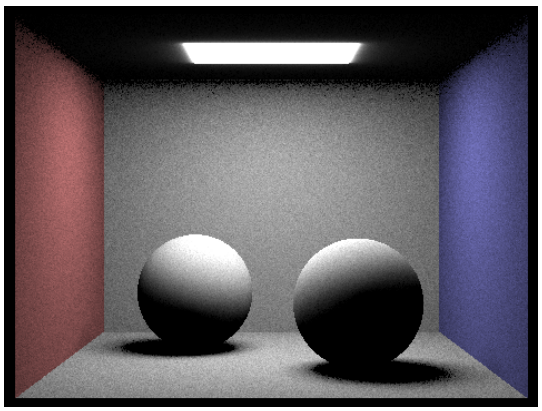
then cast a shadow ray originating from the hit_p with direction of w_j. If the shadow ray intersects a light source we add (brdf * cos_theta_j * i.bsdf->get_emission()) to approximate the integral over the hemisphere and repeat for the number of samples.

Importance Sampling

In importance sampling, for each light in the scene we sample directions between the light source and the hit_p. If the light is a point light we can store the calculations for all future point lights as point light radiance are uniform. If not we sample from the light to retreive the radiance, sampled direction, distance to the light, and probability density function. The BRDF is then calculated using the incident light direction wj and the outgoing light direction w_out. The irradiance contribution from a light source in the direction w_j is proportional to the cosine of the angle between the surface normal N and w_j which we can calculate the dot product between these two vectors. We then cast a shadow ray originating from the hit_p with direction of w_j if the light source is ahead the hit_p. If the shadow ray intersects an object and the intersection is after the light source then we are not in shadow and can add the contribution of that light. We normalize and return the calculated sum.

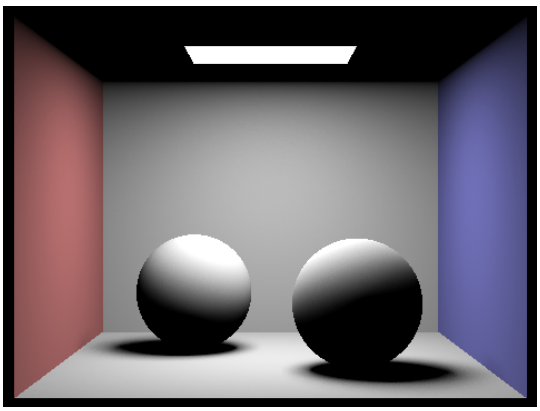
Show some images rendered with both implementations of the direct lighting function.

Uniform Hemisphere Sampling

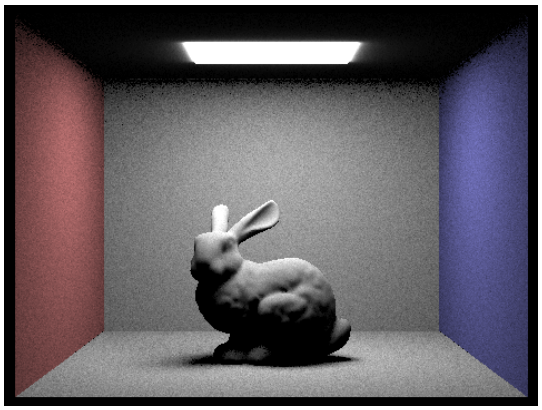


Uniform Hemisphere Sampling (CBspheres_lambertian.dae)

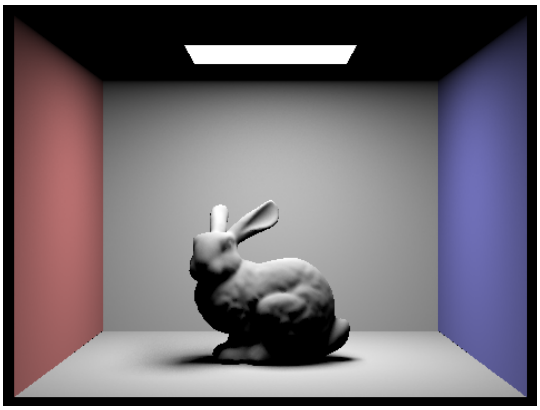
Light Sampling



Importance Sampling (CBspheres_lambertian.dae)

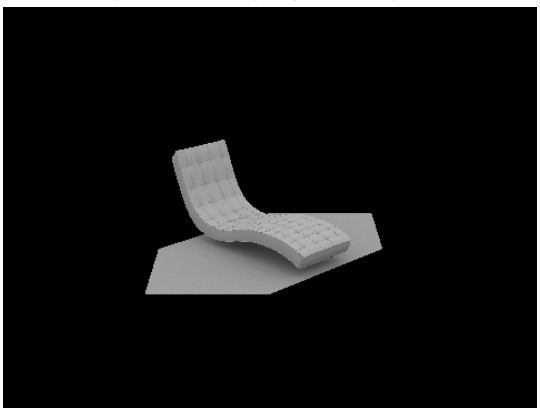


Uniform Hemisphere Sampling (CBbunny.dae)



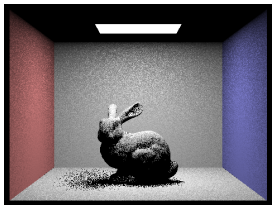
Importance Sampling (CBbunny.dae)

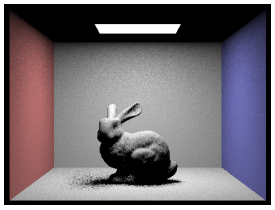
Uniform Hemisphere Sampling (bench.dae)

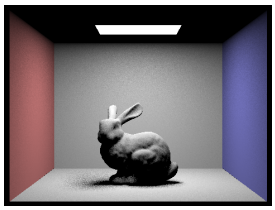


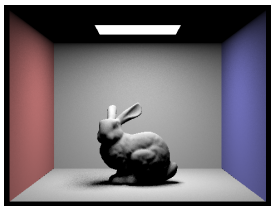
Importance Sampling of (bench.dae)

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 of CBbunny.dae


4 Light Rays of CBbunny.dae


16 Light Rays of CBbunny.dae


64 Light Rays of CBbunny.dae

For the 1 light ray image we can see the very large and apparent amount of noise near the bottom of the bunny's shadow as there aren't enough samples per light. Increasing the number of light rays to 4 provides a bit more detail and less noise but the shadows aren't very soft. With 16 rays, we can see definite softness in the shadow edges and a large reduction in noise compared to both the 1 and 4 ray image. At 64 rays, the noise within the shadow region is greatly reduced and the edges of the shadows are much softer, creating a more realistic representation of the shadows.

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

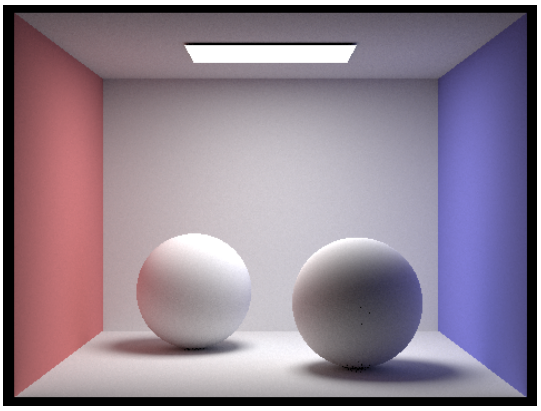
The major difference between uniform hemisphere sampling and importance sampling is that uniform hemisphere sampling exhibits a lot more noise on the walls and especially near the shadows which makes sense as any ray is equally likely to be sampled including rays that are heavily attenuated near the horizon causing noise. However in importance lighting sampling we sample directly from light sources, thus giving us a less noisy image. Additionally, point light sources cannot be hemisphere sampled as the probability of picking a single point in a PDF is 0 which is why on the left we see a black image for the bench. Importance sampling from light sources allows us to render point light sources correctly.

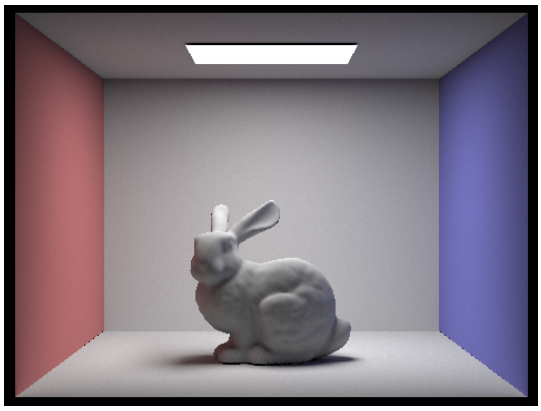
Part 4: Global Illumination (20 Points)

Walk through your implementation of the indirect lighting function.

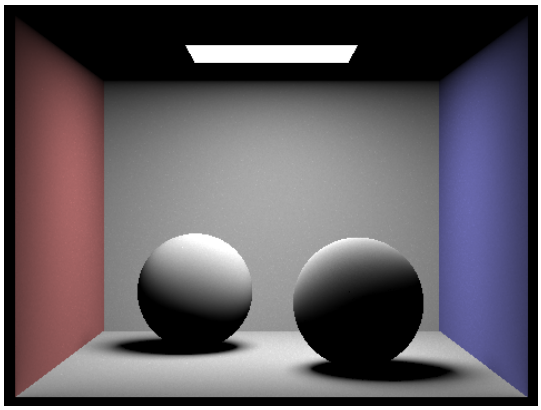
To implement the indirect lighting function, we first sample an incident light direction and the pdf of that sampled direction. Using the incoming wi and outgoing w_out light directions, we compute the BRDF. We then transform the sampled direction to world space and cast a shadow ray from the intersection point, decreasing its depth by one. If the shadow ray intersects an object and we are accumulating bounces we accumulate the indirect illumination by adding to L_out and make a recursive call to the indirect lighting function using the shadow ray and shadow intersection. However, if we are not accumulating bounces and the shadow ray's depth has reached zero, we return this mth bounce and return the radiance given by the one_bounce_radiance function. If russian roulette is used, we determine whether to terminate the ray's path via a coin flip function with p = 0.3 and if the path continues we must scale by the continution probability.

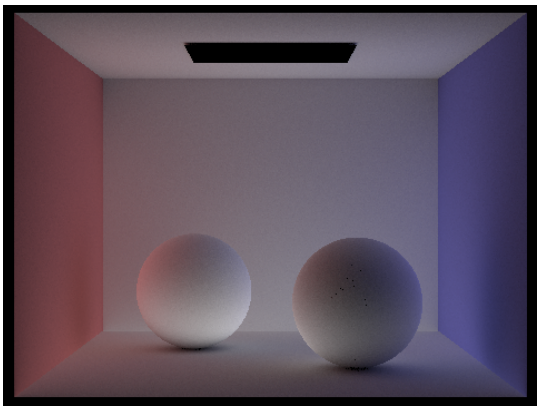
Show some images rendered with global (direct and indirect) illumination. Use 1024 samples per pixel.


Global Illumination on Spheres using 5 bounces


Global Illumination on a bunny using 2 bounces

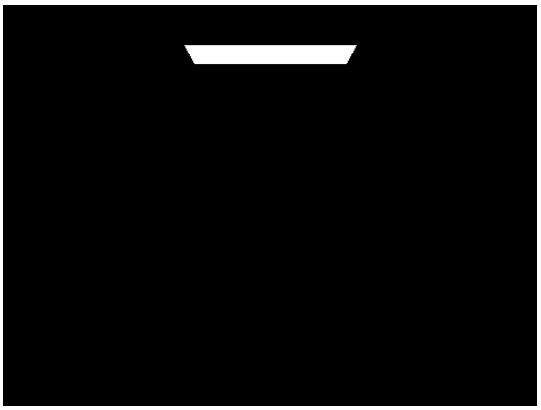
Pick one scene and compare rendered views first with only direct illumination, then only indirect illumination. Use 1024 samples per pixel. (You will have to edit PathTracer::at_least_one_bounce_radiance(...) in your code to generate these views.)

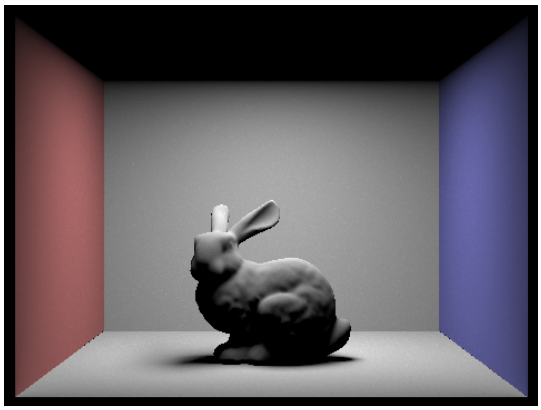

Only direct illumination (CBspheres_lambertian.dae)

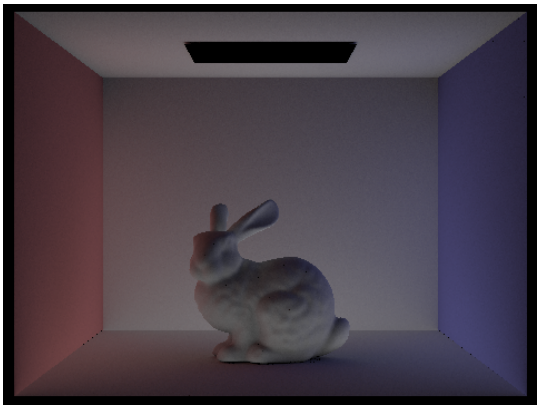

Only indirect illumination (CBspheres_lambertian.dae)

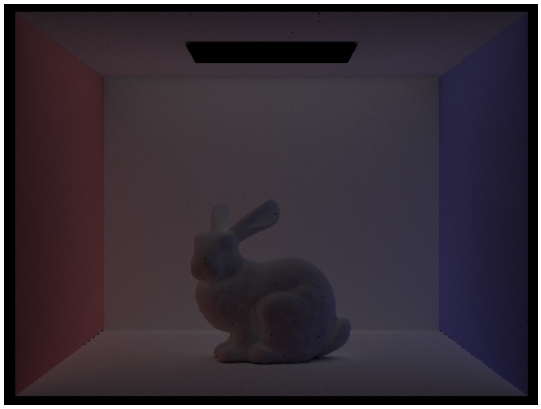
When comparing Direct Illumination which encompasses only zero + one bounce illumination, with Indirect Illumination, all illumination provided by more than one bounce we can see that Direct Illumination results in much more pronounced shadows due to substantial impact of the initial light rays. This also leads to much darker areas where light must need more than one bounce to reach such as the ceiling. For the indirect illumination image, we can see no effect from the zero bounce light rays which is why the area light is essenitally turned off. We also notice the scene is much more uniformly lit and dimmer as the later light rays are much less intense.

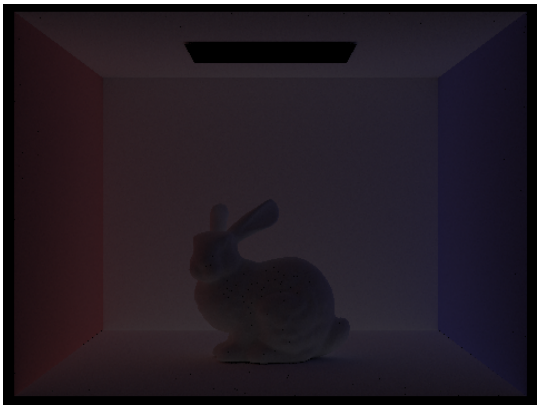
For CBbunny.dae, render the mth bounce of light with max_ray_depth set to 0, 1, 2, 3, 4, and 5 (the -m flag), and isAccumBounces=false. Explain in your writeup what you see for the 2nd and 3rd bounce of light, and how it contributes to the quality of the rendered image compared to rasterization. Use 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)

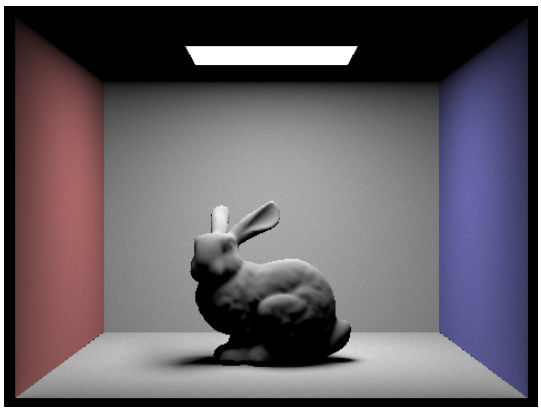

max_ray_depth = 4 (CBbunny.dae)

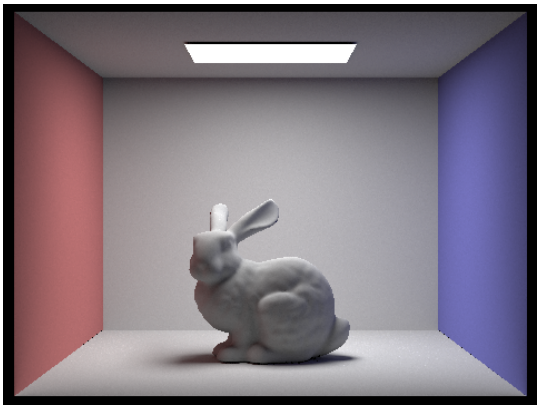

max_ray_depth = 5 (CBbunny.dae)

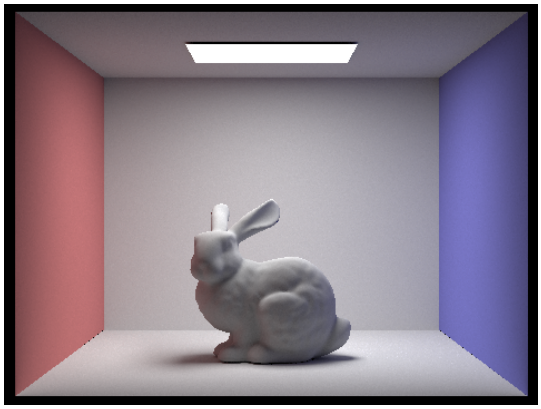
In the rendered images, the second bounce of light is visible as soft shadows and color bleeding on the rabbit, indicating light reflecting from the colored walls and floor. The third bounce contributes to the nuanced global illumination in corners and subtle light gradations across the rabbit and the room, enhancing the depth and realism. These bounces add layers of complexity and detail that rasterization cannot replicate as rasterization calculates the color of each pixel based on direct light sources and simple shading models without simulating the physical interactions of light rays in an environment.

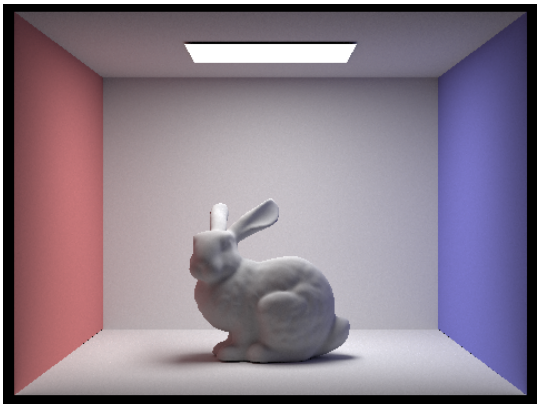
For CBbunny.dae, compare rendered views with max_ray_depth set to 0, 1, 2, 3, 4, and 5 (the -m flag). Use 1024 samples per pixel.

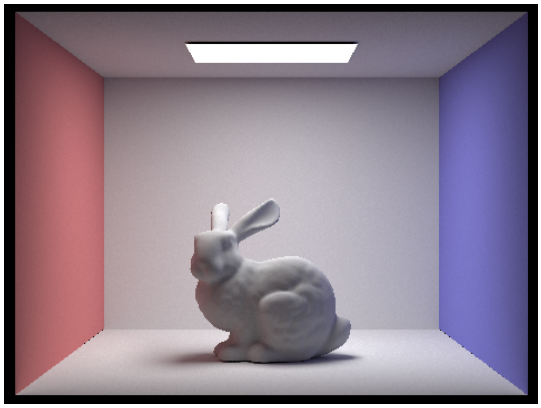

max_ray_depth = 0 (CBbunny.dae)


max_ray_depth = 1 (CBbunny.dae)


max_ray_depth = 2 (CBbunny.dae)

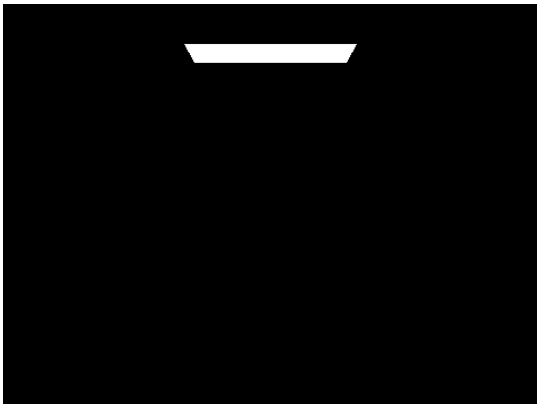

max_ray_depth = 3 (CBbunny.dae)

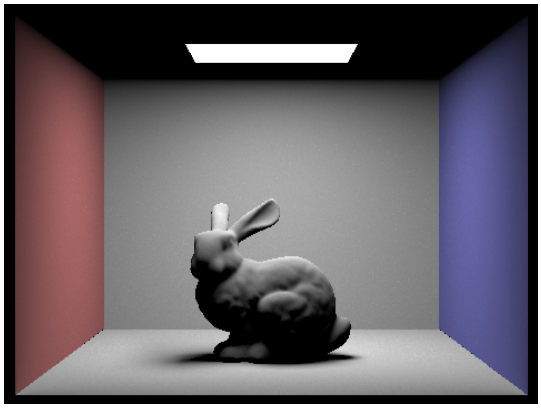

max_ray_depth = 4 (CBbunny.dae)

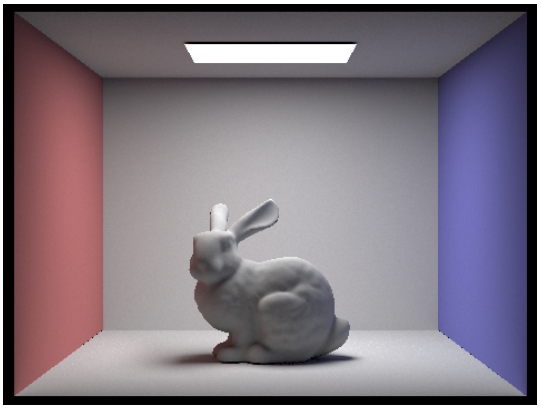

max_ray_depth = 5 (CBbunny.dae)

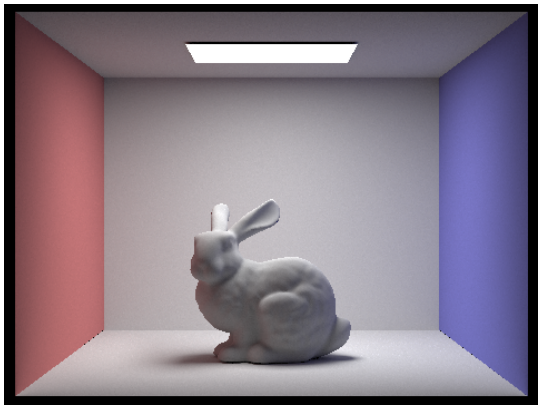
Comparing images with max_ray_depth equal to 0, 1, 2, 3, 4, and 5, we can see that as we increase the max ray depth the image becomes brighter. With zero bounce illumination, we can see that only rays that arrive directly from the light source are included which is why only the area light shines. With one bounce, we illuminate the scene but we can still see that the ceiling is not lit as it takes two bounces of light to shine the ceiling. As we start to add more indirect illumination, the scene becomes brighter and brighter and the shadows start to show more of a penumbra.

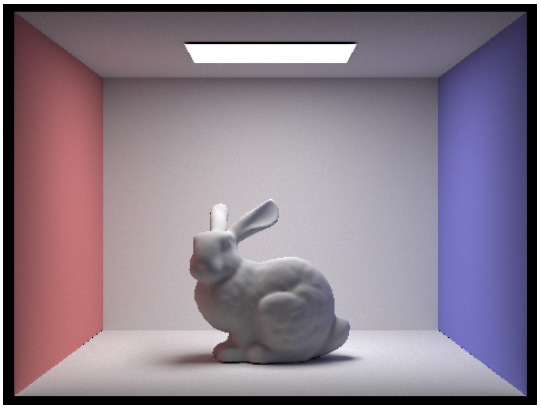
For CBbunny.dae, output the Russian Roulette rendering with max_ray_depth set to 0, 1, 2, 3, 4, and 100(the -m flag). Use 1024 samples per pixel.

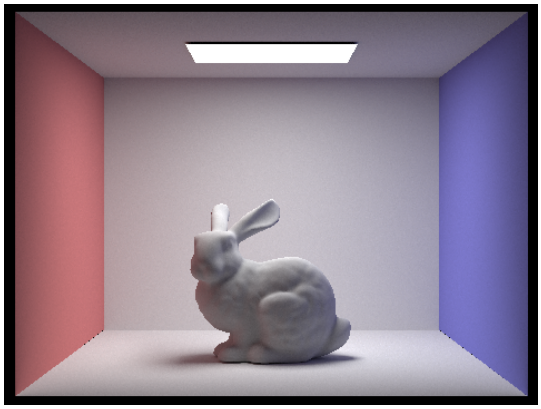

max_ray_depth = 0 (CBbunny.dae)


max_ray_depth = 1 (CBbunny.dae)

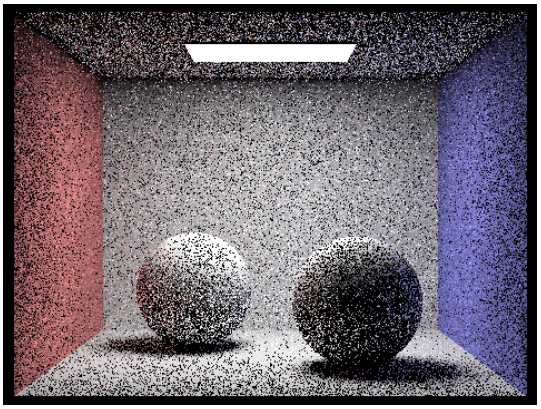

max_ray_depth = 2 (CBbunny.dae)

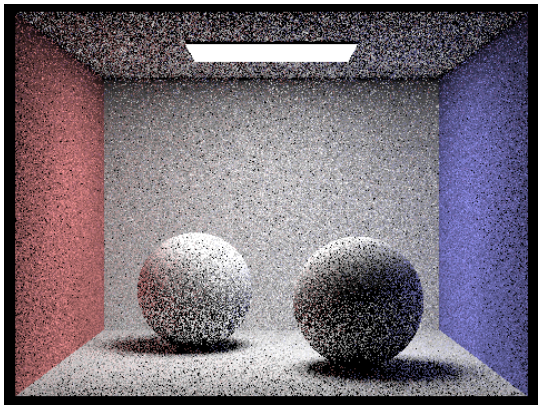

max_ray_depth = 3 (CBbunny.dae)

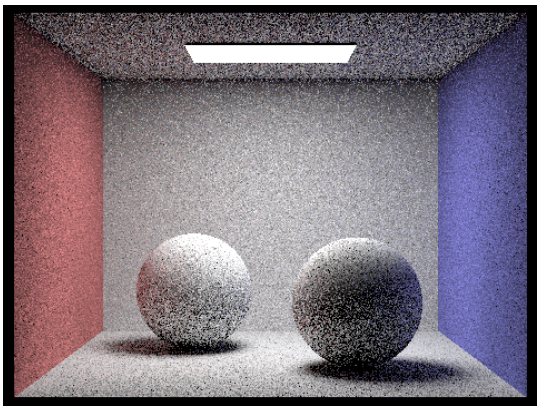

max_ray_depth = 4 (CBbunny.dae)

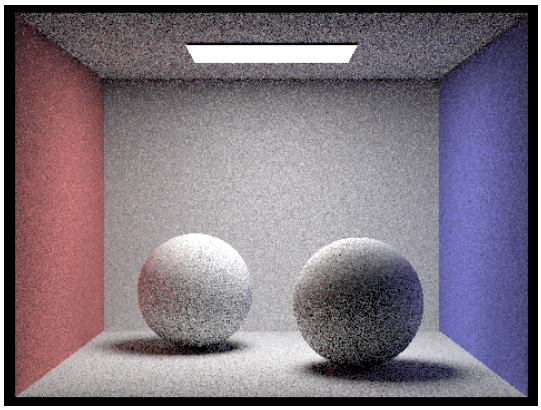

max_ray_depth = 100 (CBbunny.dae)

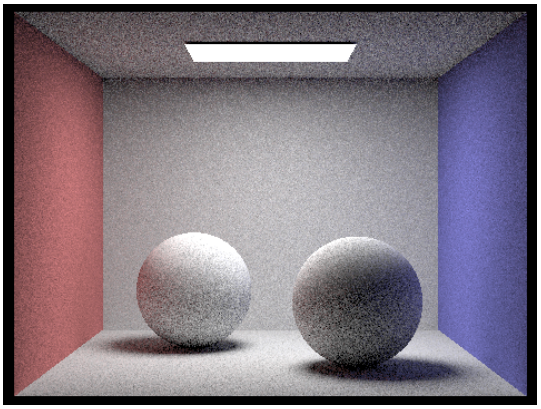
Pick one scene and compare rendered views with various sample-per-pixel rates, including at least 1, 2, 4, 8, 16, 64, and 1024. Use 4 light rays.

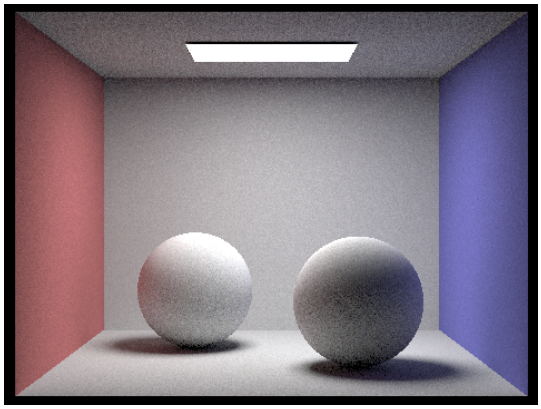

1 sample per pixel (CBspheres_lambertian.dae)

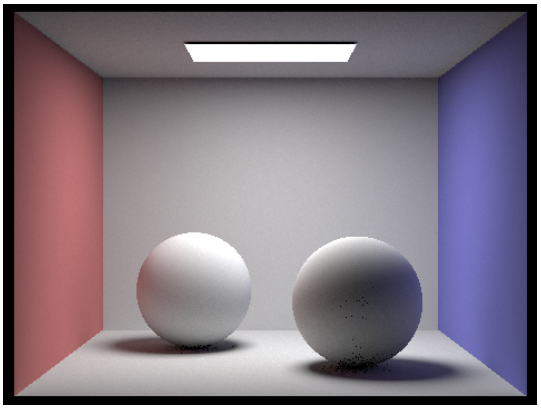

2 samples per pixel (CBspheres_lambertian.dae)


4 samples per pixel (CBspheres_lambertian.dae)


8 samples per pixel (CBspheres_lambertian.dae)


16 samples per pixel (CBspheres_lambertian.dae)


64 samples per pixel (CBspheres_lambertian.dae)


1024 samples per pixel (CBspheres_lambertian.dae)

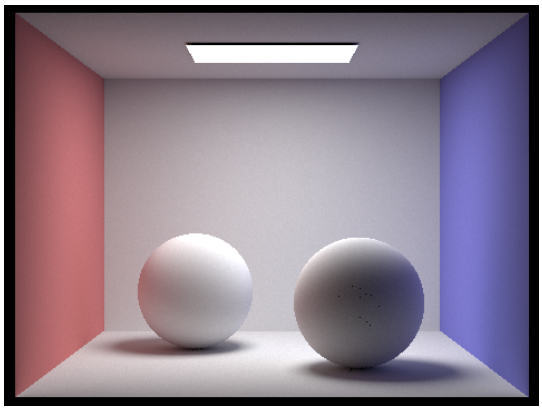
When comparing different per pixel sample rates, we can clearly see as we increase the sample rate, the amount of noise in the image decreases. We can see the large amount of noise when we sample at 1, 2, or 4 samples per pixel and with a much higher sample rate in 1024 samples we can see the apparent lack of noise.

Part 5: Adaptive Sampling (20 Points)

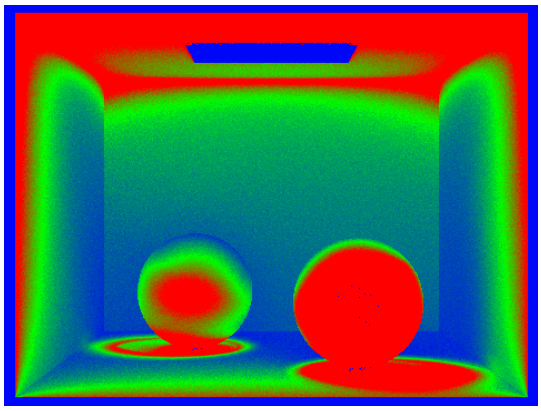
Explain adaptive sampling. Walk through your implementation of the adaptive sampling.

Adaptive sampling is a technique to create noise-free images by increasing the per pixel sample rate and concentrating the pixels on more difficult parts of the image. My implementation of adaptive sampling consisted of keeping track of two sums for the pixel s1 and s2, one for the sample illuminance and one for the sample illuminance squared. Every samplesPerBatch I calculate the mean u of the current n samples by dividing s1 / n and the variance o^2 of the current n samples through `(1 / (n – 1)) * (s2 * (s1**2)/n)`. I then `I = 1.96 * sqrt(o^2)/sqrt(n)` and if I <= maxTolerance * u, we can stop generate rays and assume the pixel has converged. We then must divide the calcualted radiance by the number of samples taken.

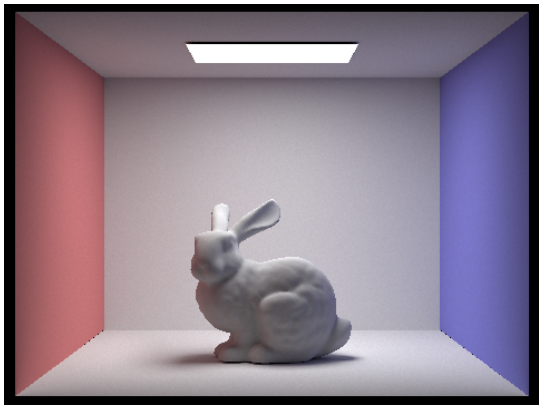
Pick two scenes and render them with at least 2048 samples per pixel. Show a good sampling rate image with clearly visible differences in sampling rate over various regions and pixels. Include both your sample rate image, which shows your how your adaptive sampling changes depending on which part of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



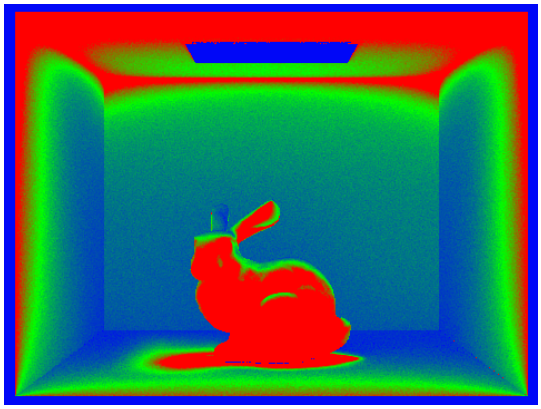
Rendered image of spheres (CBspheres_lambertian.dae)



Sample rate image of spheres (CBspheres_lambertian.dae)



Rendered image of a bunny (CBbunny.dae)



Sample rate image of a bunny (CBbunny.dae)