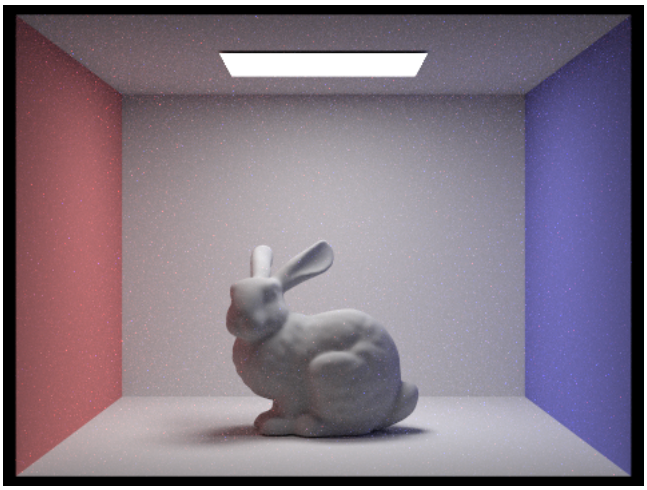
CS 184: Computer Graphics and Imaging, Spring 2024

Homework 3: PathTracer

Ian Dong and Colin Steidtmann

Overview

In this homework, we implemented a path tracing renderer. First, we worked on generating camera rays from image space to sensor in camera space and their intersection with triangles and spheres. Then, we built a bounding volume hierarchy to accelerate ray intersection tests and speed up the path tracers rendering. Afterwards, we explored direct illumination to simulate light sources and render images with realistic shadowing. Then, we implemented global illumination to simulate indirect lighting and reflections using diffuse BSDF. Finally, we implemented adaptive sampling to reduce noise in the rendered images.



My bunny is the bounciest bunny

Section I: Ray Generation and Scene Intersection (20 Points)

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

For the ray generation portion of the rendering pipeline, we first made sure to find the boundaries of the camera space by calculating $\tan(\frac{\text{hFov}}{2})$ and $\tan(\frac{\text{vFov}}{2})$ since the bottom left corner is defined as $(-\tan(\frac{\text{hFov}}{2}), \tan(\frac{\text{vFov}}{2}) - 1)$ and the top right corner is defined as $(\tan(\frac{\text{hFov}}{2}), \tan(\frac{\text{vFov}}{2}), -1)$. Then, we used the instance variables `hFov` and `vFov` which are in degrees to calculate the height and width length before using linear interpolation to find the camera image coordinates. Afterwards, we used `this->c2w` to convert my camera image coordinates into world space coordinates and also normalized the direction vector. Finally, we constructed the ray with this vector and defined the `min_t` and `max_t`.

For the primitive intersection portion of the rendering pipeline, we generated `num_samples` using `this->gridSampler->get_sample()`. We made sure to normalize the coordinates before calling on the previously implemented method to generate

the ray. Finally, we called `this->est_radiance_global_illumination()` to get the sample radiance and averaged the radiance to update the pixel in the buffer.

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

For the ray-triangle intersection, we implemented the Moller-Trumbore formula. This algorithm takes in a ray with origin, o, and direction, d, as well as a triangle with vertices, p_0, p_1, and p_2 and solves the following equation:

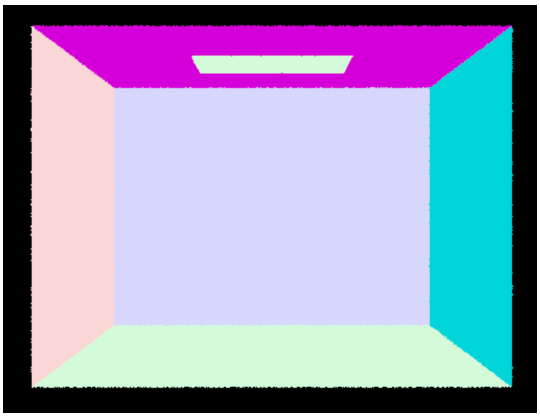
$$\vec{O} + t\vec{D} = (1 - b_1 - b_2)\vec{p}_0 + b_1\vec{p}_1.$$

We followed the algorithm by defining each of the variables and solving for t, b_1, and b_2. If t was not within the range of the minimum and maximum time range, the ray would be parallel to the triangle and thus would not intersect with the triangle given this time range. Otherwise, the ray would intersect within the triangle's plane. However, we needed to make sure it was within the triangle so we checked the barycentric coordinates to ensure they were both within [0, 1]. If they were, we updated the intersection struct.

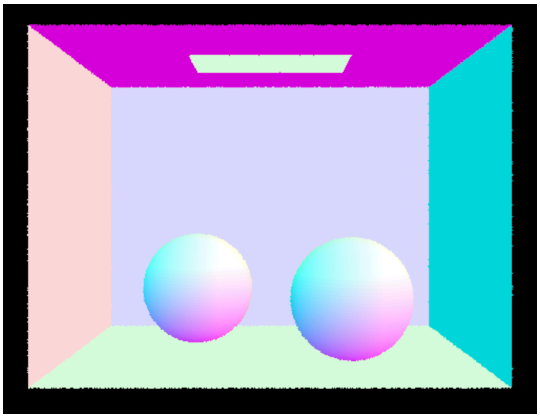
For the ray-sphere intersection, we followed the steps in the class slides. We set the equation of the ray equal to the equation of the sphere and solved for the intersection with the quadratic formula. We checked to see if the discriminant was positive so that we could find the times of intersection. Because it was a quadratic equation, there could be up to two solutions and assigned the smaller one to `t1` and the larger one to `t2`. If these times of intersection were within the ray's time range, we updated the intersection struct.

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

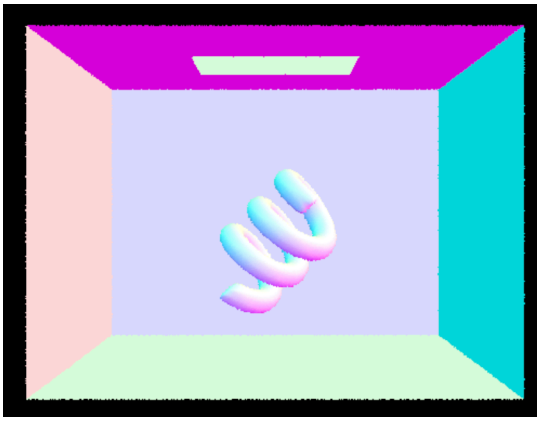
Here are some screenshots of the .dae files rendered with normal shading:



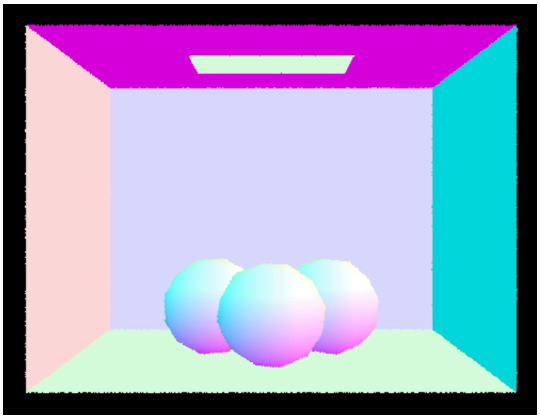
CBempty.dae



CBspheres.dae



CBcoil.dae



CBgems.dae

Section II: Bounding Volume Hierarchy (20 Points)

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

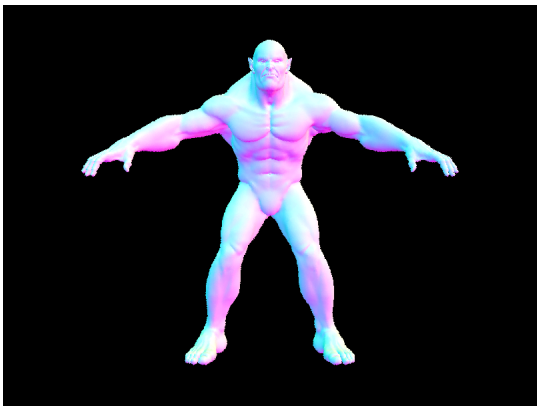
We implemented a recursive BVH construction algorithm. These were the formal steps and cases.

1. Base Case: If the number of primitives is less than or equal to `max_leaf_size`, then we created a leaf node and assigned its start and end to the passed in start and end iterators. Finally, we returned this leaf node.
2. Recursive Case: Otherwise, we needed to find the best split point to create the left and right BVH nodes. First, we iterated through all three dimensions and created a new function to find the median of the primitives for the current dimension. we temporarily split the primitives into the two nodes based on this median axis. The heuristic we used was the sum of the surface areas of the two bounding boxes and chose the axis that minimized this sum. Afterwards, we split the primitives into the two nodes, updated the iterator to connect them, and found the midpoint before passing in the new start and end iterators into the recursive BVH construction algorithm. If at any time a split led to all of the primitives being in one node, we would just follow the base case logic and assign the start and end to the node. Finally, we returned the node.

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

Here are some screenshots of the `.dae` files rendered with normal shading using BVH acceleration:

`cow.dae`



`beast.dae`

`maxplanck.dae`

`beetle.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.

As shown in the table below, we found significant speedups in rendering times when using BVH acceleration. We used three `.dae` scenes with differing number of primitives. It looks that the rendering time is proportional to the average number of intersection tests per ray. Without BVH acceleration, we had to cast every single ray on every primitive and thus it scales linearly with the number of primitives. With BVH acceleration, we split the primitives into two different nodes so effectively reduced it down logarithmically and thus do not need to check as many primitives so the intersection tests remain relatively constant. The BVH data structure helps us to quickly find the intersection of a ray with the scene and thus significantly reduces the time it takes to render the scene.

`.dae` Scene	Number of Primitives	Render Time (no BVH)	Render Time (BVH)	Avg Intersection Tests per Ray (no BVH)	Avg Intersection Tests per Ray (BVH)
`teapot.dae`	2464	43.25 s	0.0752 ms	1006.19	2.89
`peter.dae`	40018	697.63 s	0.0837 s	11753.02	2.41
`CBlucy.dae`	133796	2640.11 s	0.1127 s	30217.62	3.03

Section III: Direct Illumination (20 Points)

Walk through both implementations of the direct lighting function.

Direct lighting is zero bounce lighting, the light that comes directly from the light source, plus one bounce lighting, the light that comes back to the camera after reflecting off the scene once. For zero bounce, we only need to return the light from the light source without any bounces. However, for one bounce, we need to determine how much light is reflected back to the camera after the ray intersects with the scene. Because we cannot compute an infinite integral, we instead used a Monte-Carlo Estimator of the reflectance.

For uniform hemisphere sampling, we iterated through the number of samples and sampled a vector uniformly from the hemisphere and converted it into the world space. Afterwards, we created the ray with this vector as the direction. If the ray intersected the scene, we would calculate the BSDF $f(\text{w\_out}, \text{w\_in})$, the emitted radiance L_i, and the angle between the surface normal and the sampled vector. Finally, we computed the sample mean of the reflectance calculations from lecture using the following formula and previous calculations:

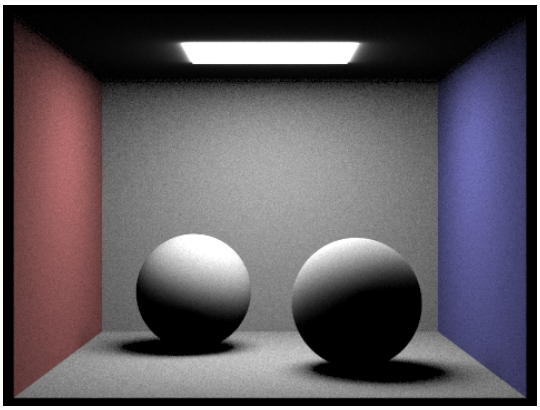
$$\frac{1}{N}\sum_{i=1}^{n}\frac{f_r(\text{p}, \omega_i \rightarrow \omega_r)L_i(\text{p}, \omega_j)\cos\theta_j}{p(\omega_j)}$$

For importance lighting sampling, instead of sampling from a uniform hemisphere we iterated through each of the light sources and calculated the number of samples needed based on if it was a delta light and sampled uniformly from each light source. Then, we iterated through the number of samples and calculated the emitted radiance along with the sampled world space vector for our ray. If the ray intersected the scene, we would calculate the BSDF and the angle between the surface normal and the sampled vector and rejected rays that were on the opposite side of the surface. For each light, we computed the mean reflectance using the formula from above. Finally, we added this mean reflectance to the total reflectance and returned the total reflectance.

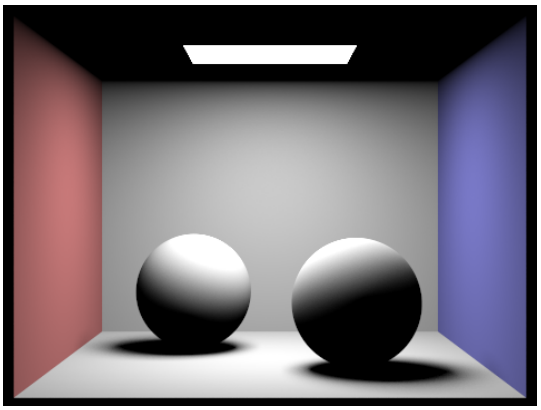
Show some images rendered with both implementations of the direct lighting function.

Here are some images rendered with both implementations of the direct lighting function:

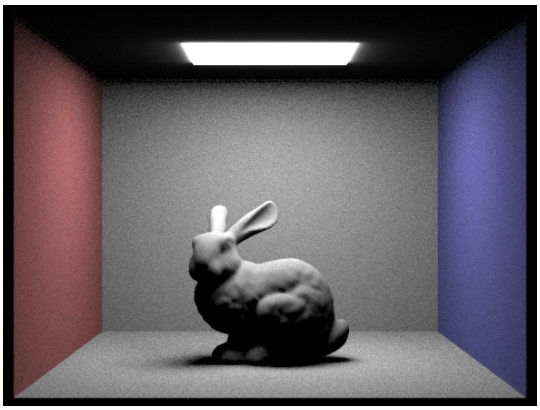
Uniform Hemisphere Sampling	Light Sampling



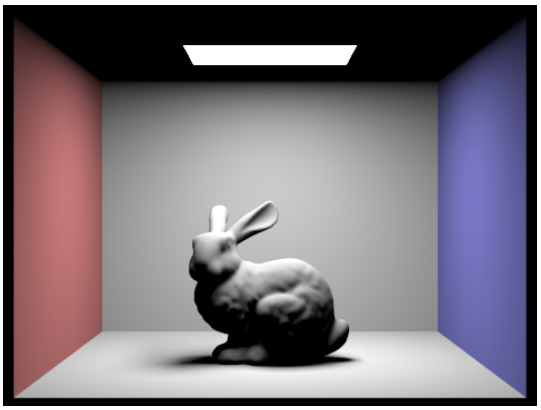
CBspheres_lambertian.dae



CBspheres_lambertian.dae



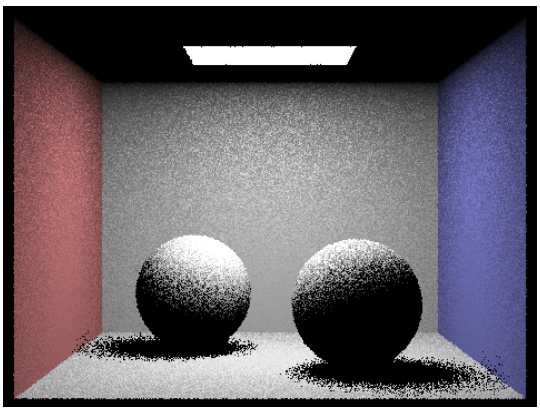
CBbunny.dae



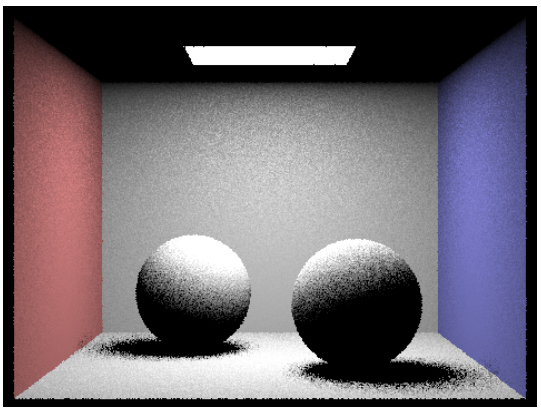
CBbunny.dae

We noticed that importance sampling converged much faster than uniform hemisphere sampling. The soft shadow noise in hemisphere sampling comes from the fact that only a small portion of the rays cast actually hit the scene. In contrast, importance lighting sampling only considers the rays that actually contribute to the illumination of the scene and thus has much less noise. This leads to much more smoother shadow scene renderings as shown in the above images.

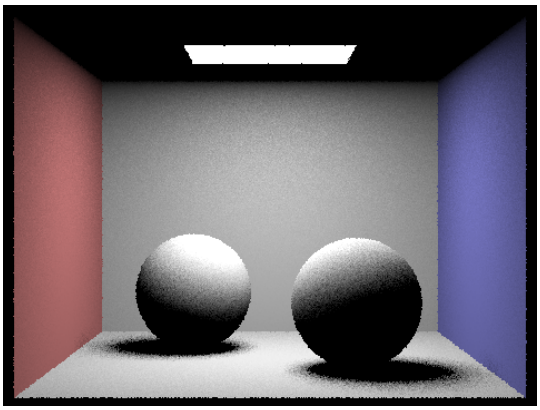
Focus on one particular scene with at least one area light and compare the noise levels in soft shadows when rendering with 1, 4, 16, and 64 light rays (the -l flag) and with 1 sample per pixel (the -s flag) using light sampling, not uniform hemisphere sampling.



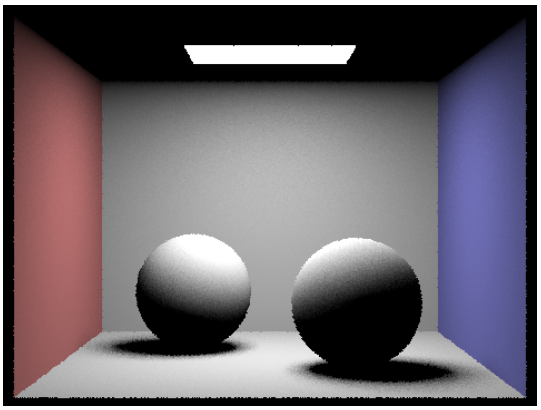
1 Light Ray (CBspheres_lambertian.dae)



4 Light Rays (CBspheres_lambertian.dae)



16 Light Rays `(CBspheres_lambertian.dae)`



64 Light Rays `(CBspheres_lambertian.dae)`

Shown in the images above, when there are low amount of light rays there are more noise in the soft shadows with individual dots making it up. As we increase the number of light rays, however, we noticed that the noise decreased dramatically. At 64 light rays, the noise was almost completely gone and the soft shadows were much smoother. This is because with more light rays, we are able to sample more points on the light source and thus get a better estimate of the light intensity at a point on the surface. This is especially important for area lights, where the light intensity can vary across the light source. Thus, the more light rays we have, the more accurate our estimate of the light intensity at a point on the surface and the smoother the soft shadows will be.

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

We noticed that importance sampling converged much faster than uniform hemisphere sampling. The soft shadow noise in hemisphere sampling comes from the fact that only a small portion of the rays cast actually hit the scene. In contrast, importance lighting sampling only considers the rays that actually contribute to the illumination of the scene and thus has much less noise. This leads to much more smoother shadow scene renderings as shown in the above images.

Section IV: Global Illumination (20 Points)

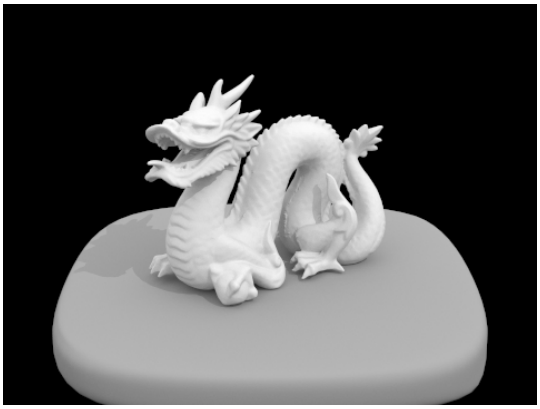
Walk through your implementation of the indirect lighting function.

We implemented a recursive function to calculate the indirect lighting as each bounce of the light will need to be calculated with the previous bounce. These were the formal steps and cases.

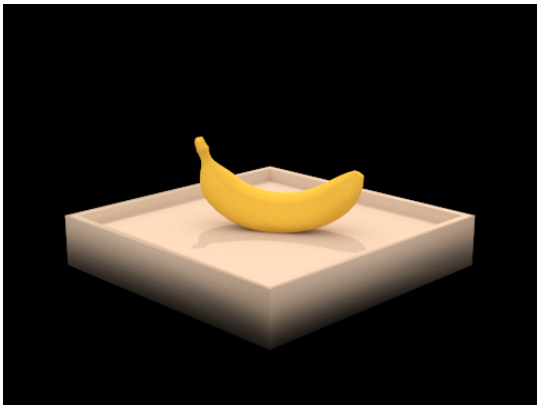
1. Base Case: If the ray's depth reaches 1, we can just return `one_bounce_radiance`.
2. Recursive Case: Otherwise, we will flip a biased coin and continue path tracing with probability continuation_prob. Then, we calculated the `one_bounce_radiance` for the current bounce. Afterwards, we sampled with the BSDF to figure out the next direction the ray will go and set the depth to be one less than the current depth. If this ray intersected with the scene, we would recurse to find the next emitted radiance and apply the reflectance formula. If `isAccumBounces` is true, we add it to the running total radiance and else we would return just the current level's radiance.

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

Here are some images rendered with global (direct and indirect) illumination:



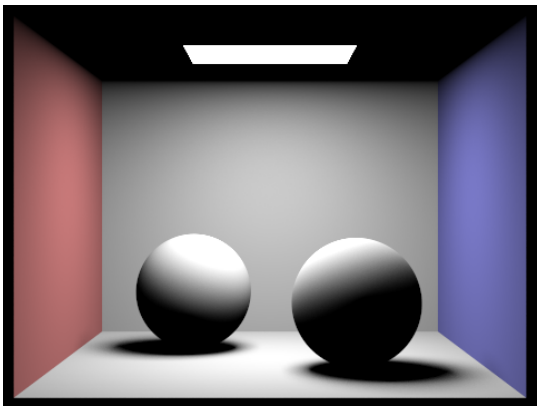
dragon.dae



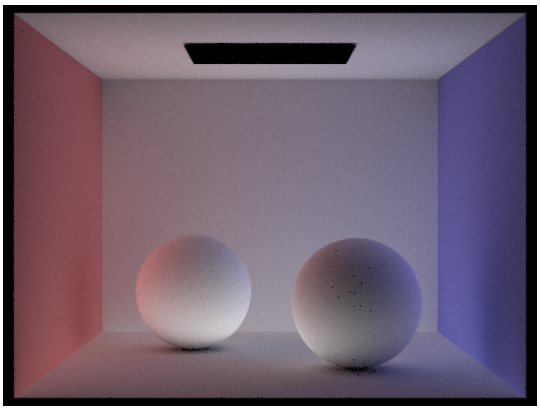
banana.dae

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.)

Here are the rendered images with only direct illumination and only indirect illumination:



Only direct illumination `(CBspheres_lambertian.dae)`

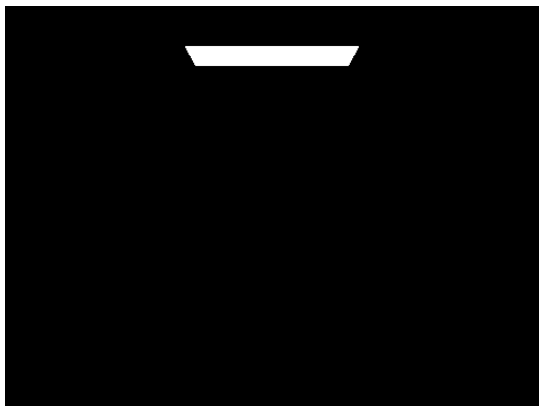


Only indirect illumination `(CBspheres_lambertian.dae)`

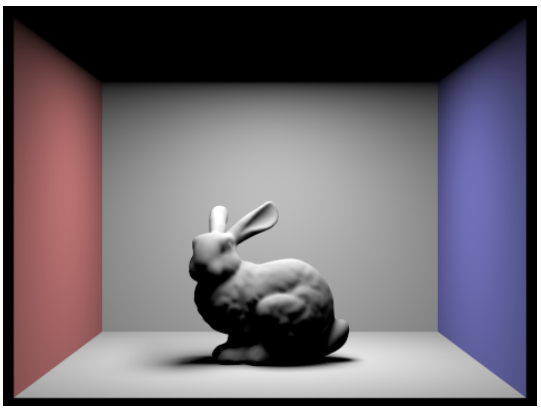
As shown above, the left scene only has direct illumination which is zero plus one bounce lighting while the right scene only has indirect illumination. Both of these illuminations present an incomplete picture. Direct illumination only illuminates the portions of the scene that the light rays can directly reach so they miss out on the the ceiling and undersides of the spheres. Indirect lighting is the opposite as it only illuminates the portions of the scene that the light rays cannot directly reach so they miss out on the the light source and the direct light and illuminate the underside of the spheres. Thus, we need both direct and indirect illumination to get a complete picture of the scene.

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.

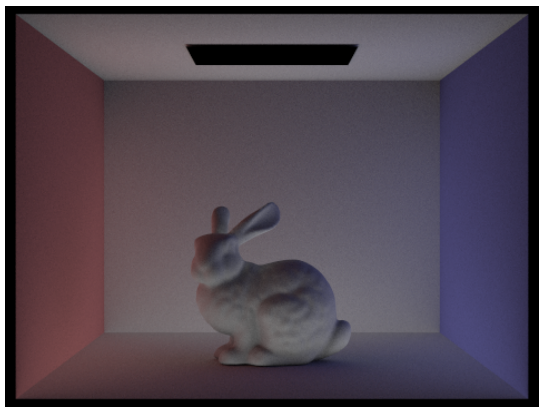
Here are the rendered images with `max_ray_depth` set to 0, 1, 2, 3, 4, and 5 and `isAccumBounces=false`:



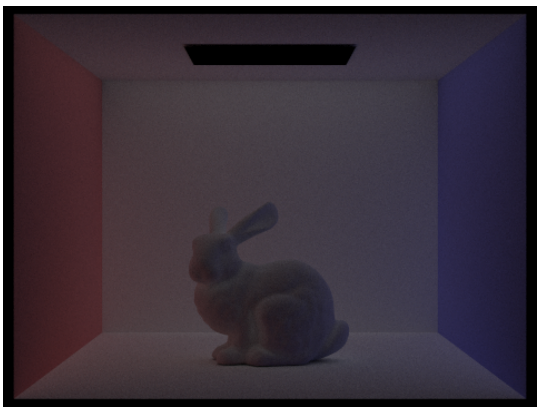
max_ray_depth=0 CBbunny.dae



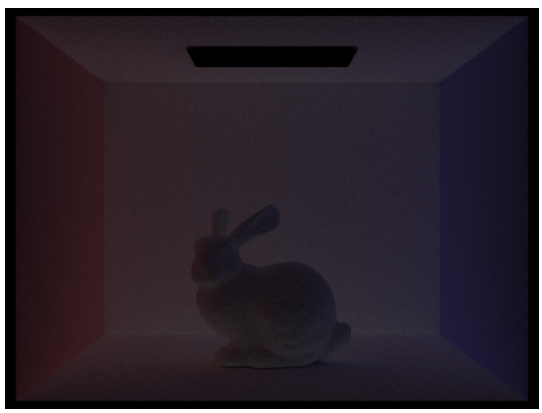
max_ray_depth=1 CBbunny.dae



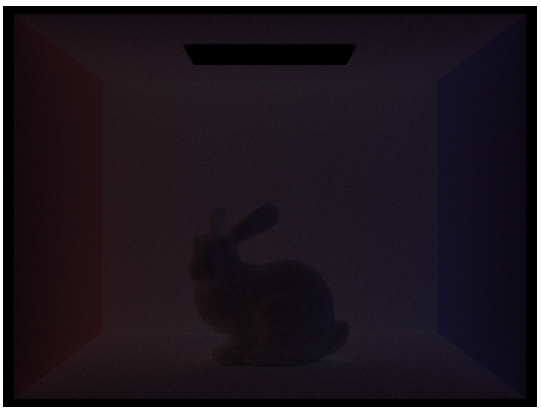
max_ray_depth=2 CBbunny.dae



max_ray_depth=3 CBbunny.dae



max_ray_depth=4 CBbunny.dae

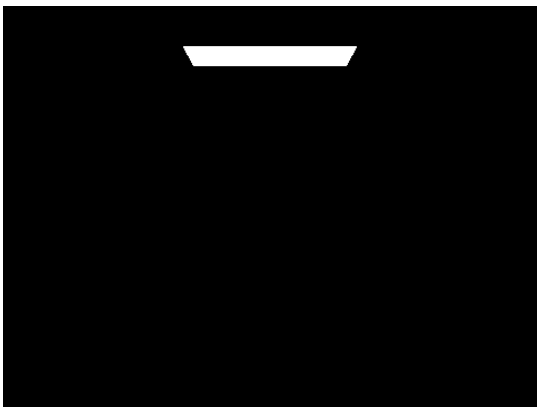


max_ray_depth=5 CBbunny.dae

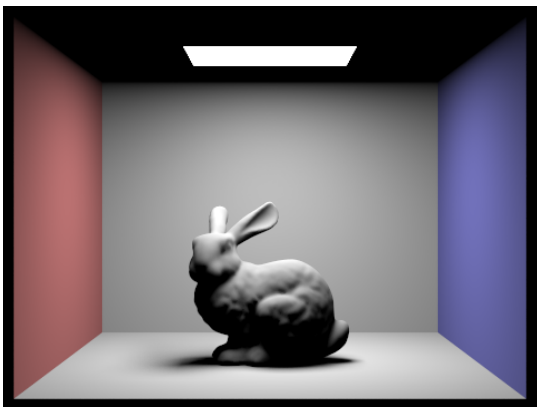
As shown above, the 2nd bounce image features a bunny with illumination on the underside of it because this light bounces off the floor and then onto the bunny. The fur on the head and back is dark because the light does not reach it on the second bounce but rather on the first bounce after it directly hits it. The 3rd bounce image is a lot more darker and the bunny is almost completely covered in shadows. This is because the the 3rd bounce is not hitting the bunny but going off to other parts of the room and is why only certain portions are illuminated. These bounce are able to contribute and help render accurate shadows, recursive reflections, and refractions which rasterization is not able to do. This helps to create a more realistic and accurate image of the scene based on the lighting.

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.

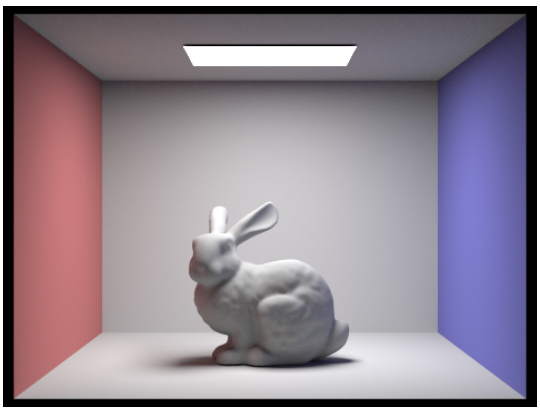
Here are the rendered images with `max_ray_depth` set to 0, 1, 2, 3, 4, and 5:



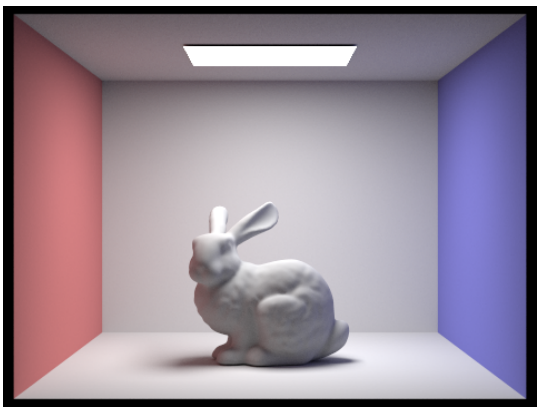
max_ray_depth=0 CBbunny.dae



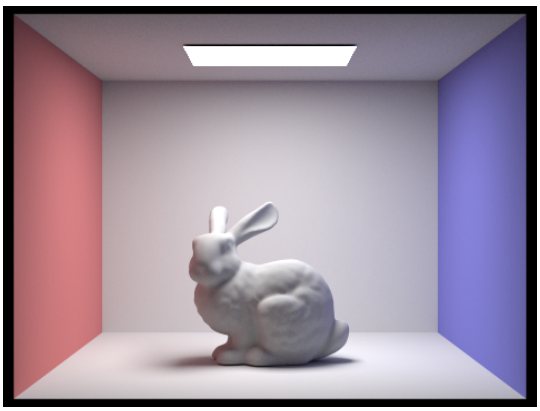
max_ray_depth=1 CBbunny.dae



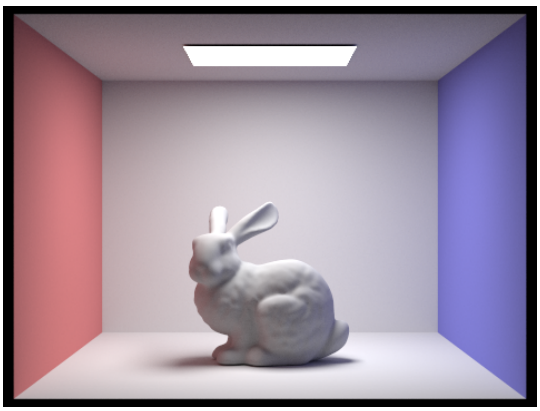
max_ray_depth=2 CBbunny.dae



max_ray_depth=3 CBbunny.dae



max_ray_depth=4 CBbunny.dae

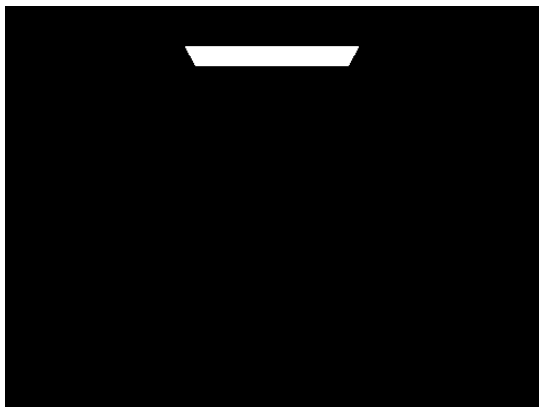


max_ray_depth=5 CBbunny.dae

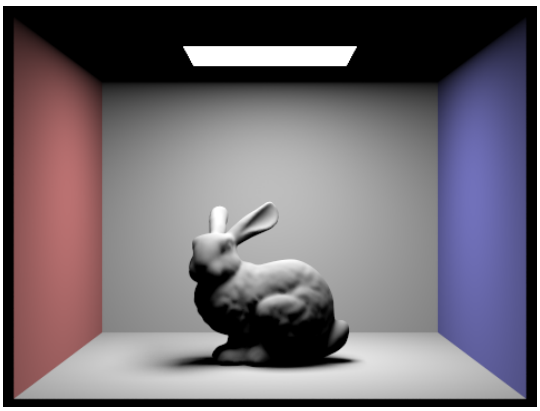
As shown in the images above, each bounce helps to convey more information about the scene as it lights up more portions. The first scene, with `max_ray_depth=0`, only has zero bounce lighting which is the light source itself. However, in the second scene with `max_ray_depth=1`, the light bounces off the floor and onto the bunny and we can observe the top of the bunny illuminated. Later bounces will also highlight the underside of the bunny as the light bounces off the floor and walls and ultimately onto the bunny. This helps to diffuse all lighting.

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.

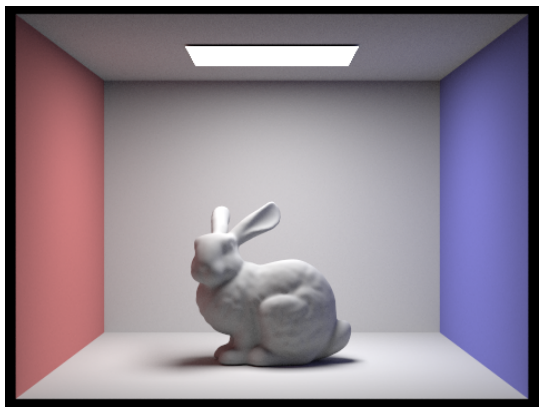
Here are the rendered Russian Roulette images with `max_ray_depth` set to 0, 1, 2, 3, 4, and 100:



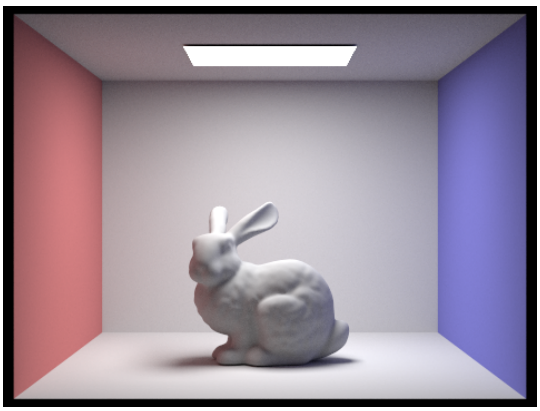
max_ray_depth=0 CBbunny.dae



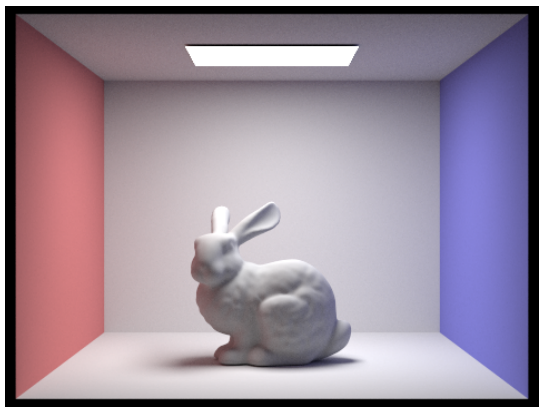
max_ray_depth=1 CBbunny.dae



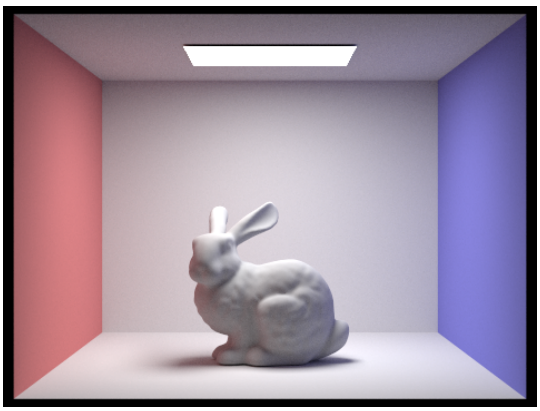
max_ray_depth=2 CBbunny.dae



max_ray_depth=3 CBbunny.dae



max_ray_depth=4 CBbunny.dae

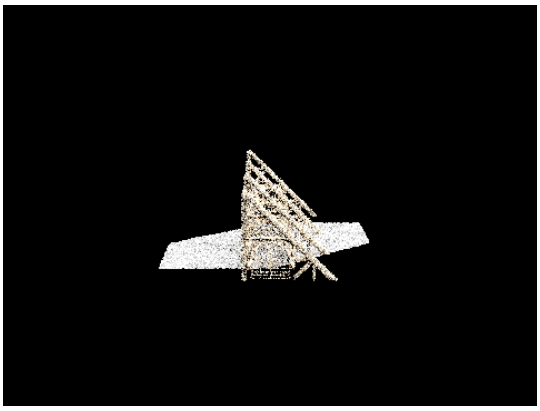


max_ray_depth=100 CBbunny.dae

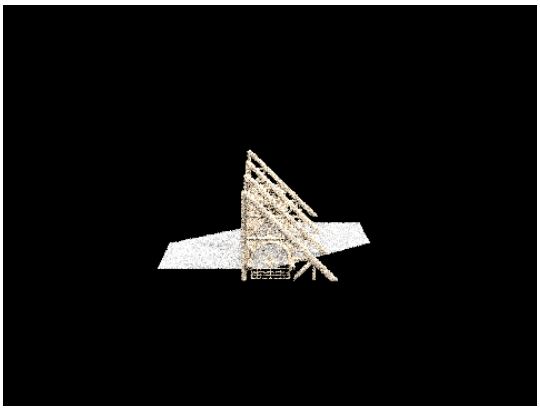
As shown in the images above, each bounce helps to convey more information about the scene as it lights up more portions. The first scene, with `max_ray_depth=0`, only has zero bounce lighting which is the light source itself. However, in the second scene with `max_ray_depth=1`, the light bounces off the floor and onto the bunny and we can observe the top of the bunny illuminated. Later bounces will also highlight the underside of the bunny as the light bounces off the floor and walls and ultimately onto the bunny. This helps to diffuse all lighting. However, at the 100 depth layer, there are not that many very large bounce rays because the continuation probability compounds and thus the rays are more likely to terminate early. Thus, it does not convey as much information as expected and looks very similar to the 4th bounce image.

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.

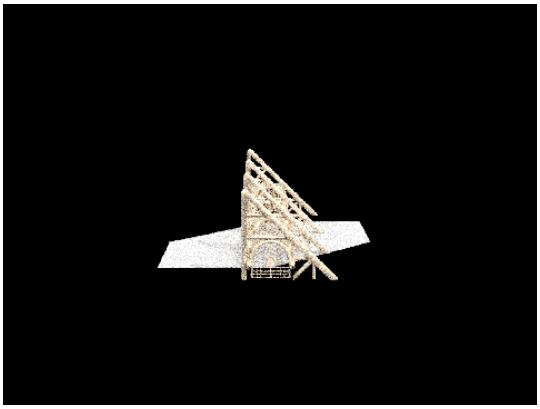
Here are the rendered images with various sample-per-pixel rates:



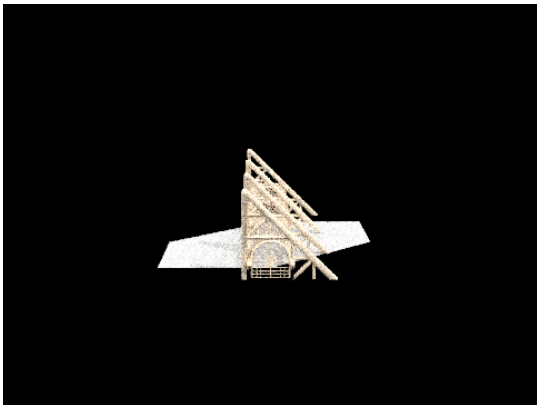
1 sample per pixel `(building.dae)`



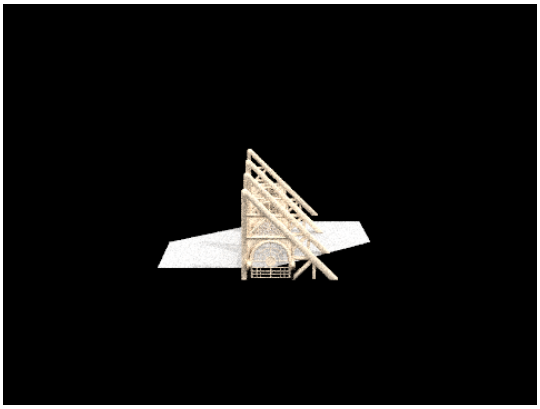
2 samples per pixel `(building.dae)`



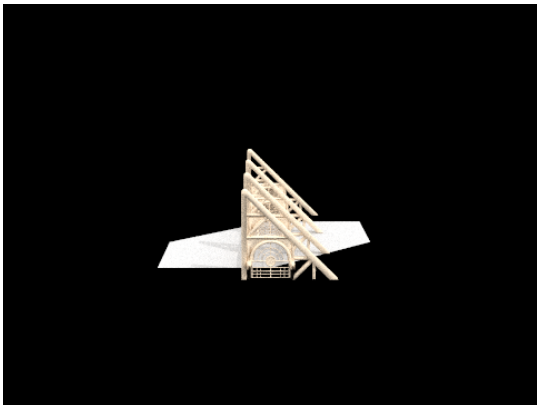
4 samples per pixel `(building.dae)`



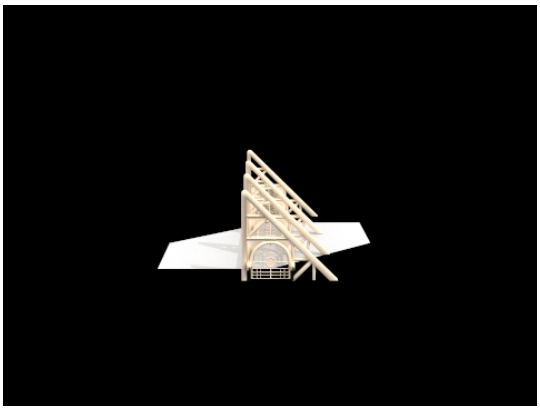
8 samples per pixel `(building.dae)`



16 samples per pixel `(building.dae)`



64 samples per pixel `(building.dae)`



1024 samples per pixel `(building.dae)`

As shown in the images above, taking more samples helps reduce the noise in the image and shadows because they become much more representative of the true scene.

Section V: Adaptive Sampling (20 Points)

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

Adaptive sampling helps to improve the efficiency and quality of rendering images, especially in situations where certain parts of the image require more computational resources to accurately represent than others. Instead of increasing the sample rate and thus the rendering time, adaptive sampling concentrates the samples in the more difficult parts of the image as there are some pixels that converge quicker than others. The idea is to allocate more samples to regions that require higher fidelity representation, while reducing the number of samples in smoother areas where details are less noticeable. We implemented adaptive sampling by updating s_1 and s_2 as defined in the spec. After a multiple of `samplesPerBatch`, we calculated the mean, standard deviation, and I. If $I \leq \text{maxTolerance} \cdot \mu$, we would stop sampling the pixel and save the total number of samples that we have taken to calculate the color correctly. We used the following equations:

$$s_1 = \sum_{i=1}^{n} x_i$$

$$s_2 = \sum_{i=1}^{n} x_i^2$$

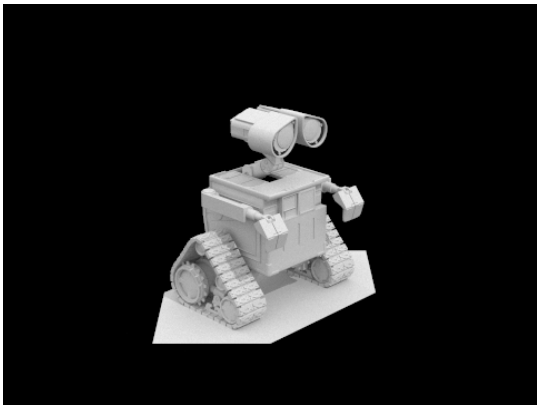
$$\mu = \frac{s_1}{n}$$

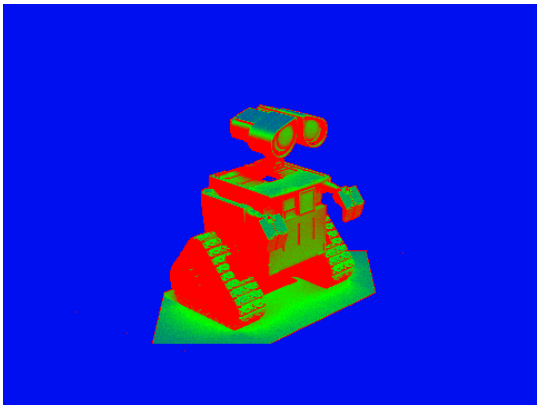
$$\sigma^2 = \frac{1}{n-1} \cdot \left(s_2 - \frac{s_1^2}{n} \right)$$

$$I = 1.96 \cdot \frac{\sigma}{\sqrt{n}}$$

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.

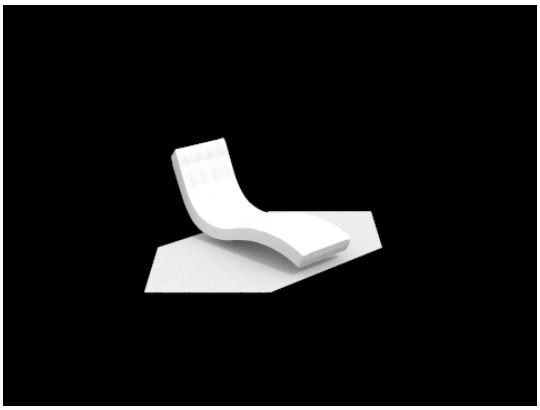
Here are the rendered and their adaptive sampled sample rate images:



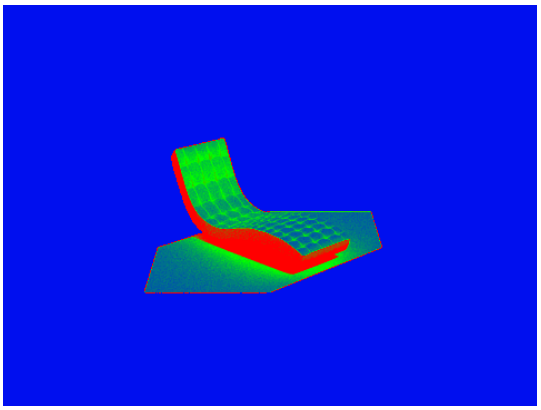


Rendered image `(wall-e.dae)`

Sample rate image `(wall-e.dae)`



Rendered image `(bench.dae)`



Sample rate image `(bench.dae)`