CS 184: Computer Graphics and Imaging, Spring 2022

Project 3

Jasmine Wang, 3034027325

Victor Ho, 3033712816

website link:

https://cal-cs184-student.github.io/sp22-project-webpages-jwang99/proj3-1/index.html

Assignment 3: PathTracer

In this Project, we created the foundation for shading and coloring objects in a lit scene through ray tracing. We first established the basic tools such as detecting intersection with primitives, mapping world points to pixel locations, and building a BVH datastructure that makes intersection detection more efficient. Then we implemented progressively more complex illumination schemes starting with direct (zero and single bounce) lighting and eventually incorporating indirect (atleast one bounce) illumination, which used the primitives previously established to determine the proper illumination for pixels representing different points in a scene. Finally, we added an algorithm to speedup the rendering process of these scenes by establishing a "sampled enough" heuristic through adaptive sampling

Part 1: Ray Generation and Scene Intersection:

Ray generation and primitve intersections

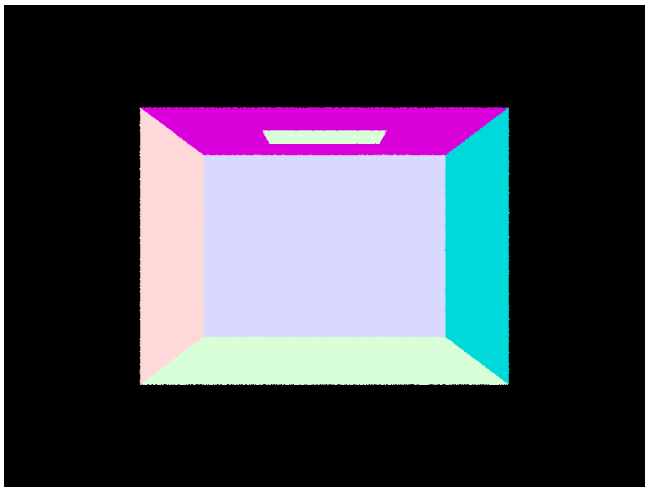
Ray generation and primitive intersection is the first step in the rendering pipeline. In order for a camera to be able to render an image of what a 3D scene looks like, it must determine what should be represented in each pixel of the image. To do so, a transfomration is first established between the camera's image (normalized image coordinates) and the camera's virtual sensor (world frame coordinates), a plane orthogonal to the camera's direction. Then, we create a ray in the the world frame, from the cameras location, to the point in the world frame which corresponds to the location of that pixel. This ray represents the "viewing direction" for the objects that should be seen in the pixel. This process is known as ray generation.

Once the appropriate ray has been generated, to properly determine what color or object should appear in the image at each image coordinate, just follow along the direction of the ray until it intersects an object primitive (triangle, sphere, etc) that is used to build objects in the scene. Once an intersection is detected, the color of the image coordinate, should be dependent on the illuminance of the object in reverse direction of the ray.

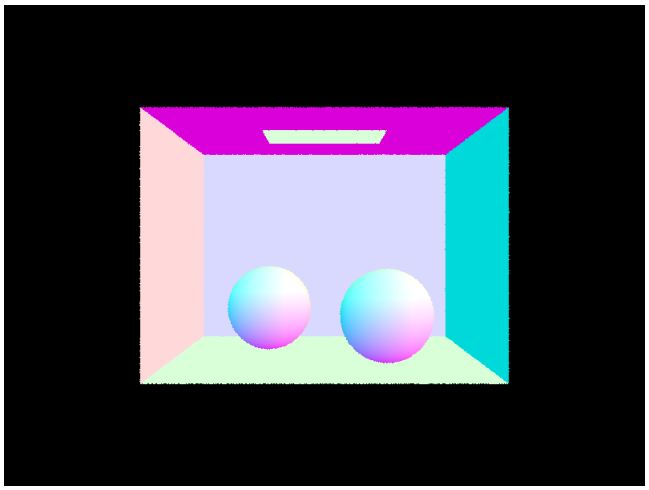
Triangle Intersection Algorithm

My triangle-ray intersection algorithm is primarily built off of the Moller Trumbore Algorithm. Using the Moller Trumbore Algorithm on the triangle-ray pair that we are finding an intersection for, we get the values t (time of intersection), b1, and b2 (two of the 3 barycentric coordinates for the triangle). We first inspect the t value to make sure that it is within the lifespan of the ray (between t_min and t_max), since rays are considered as finite segments of the entire ray. Then we check the barycentric coordinates b0 = 1-b1-b2, b1, and b2 returned by the Moller Trumbore Algorithm to make sure that they represent a point that is actually INSIDE the triangle, meaning that all barycentric coordinates are between 0 and 1. If both conditions are true, the intersection is within the lifespan of the ray and the intersection point is inside the triangle, then there is an intersection between the given ray and triangle.

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



CBEmpty.dae renderred with normal shading



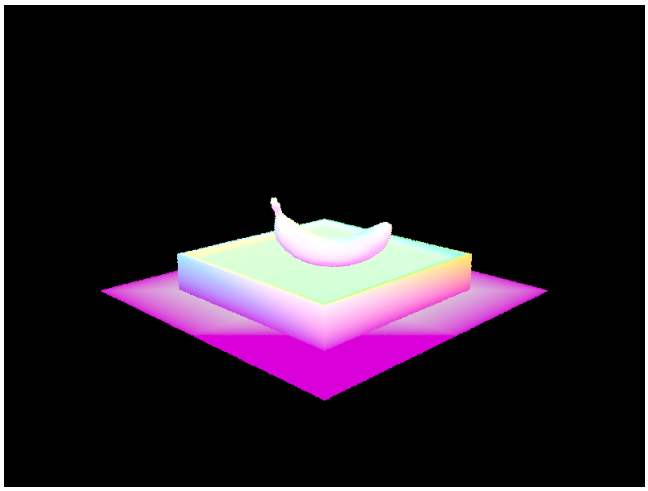
CBSpheres_lambertian.dae renderred with normal shading

Part 2: Bounding Volume Hierarchy:

BVH Construction Algorithm

The Construction of a Bounding Volume Hierarchy is a recursive process, where the BVH itself is a tree structure of BVH Nodes. When given a list of primitives that we are trying to segment into separate bounding volumes, if the number of primitives in this list is small enough to be contained in a single leaf node, then allocate a leaf node and the process is finished. Otherwise, if the number of primitives is too large to be contained in a single leaf node, we need to divide the primitives in half and create a sub-BVH tree for each half. The way that the primitives are allocated into 2 groups is: determine which axis (x, y, z) has the widest range in which primitives exist, and find the midpoint of this range. All primitives whose centroid is on one side of this midpoint gets put into one group, and all primitives whose centroid is on the other side is put into the other group. If there arises a situation in which one of these 2 groups has no primitives, forcibly move one primitve from the other group into this group.

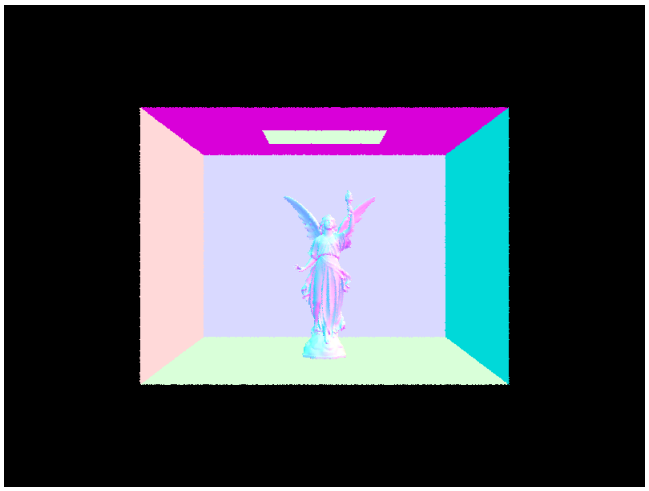
Example images with normal shading for a few large .dae files using BVH acceleration



banana.dae renderred with normal shading

cow.dae renderred with normal shading

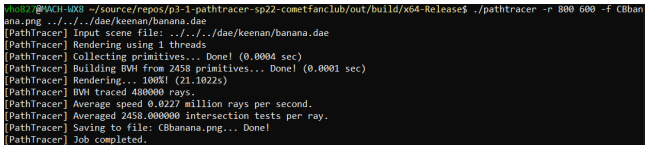
maxplanck.dae renderred with normal shading



CBlucy.dae renderred with normal shading

Render Time Comparison on Complex Geometries With and Without BVH acceleration

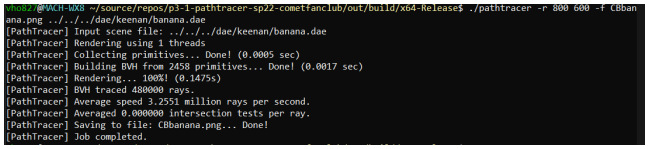
Rendering time for complex geometries decreased SUBSTANTIALLY when BVH acceleration was used. This is because we no longer have to check for intersection between each ray and EVERY SINGLE primitive in the scene we are trying to render. For complex geometries like maxplanck.dae and CBlucy.dae, these contain over 50,000 and 130,000 primitives respectively. To do a linear search for intersection with each primitive for each ray would take incredibly long. When BVH acceleration is used, not only can we first check for possible intersection with a collection of primitives by using BBox-es that envelop multiple primitives, but due to the tree structure of the BVH, we can cut down the search space of our intersetion in half with every recursive call leading to significantly improved performance. These performance improvements can be seen in the render time comparisons of the following scenes with and without BVH acceleration. Notice that CBlucy.dae was not attempted without BVH acceleration because maxplanck already couldn't finish renderring within 2 minutes despite being able to finish in .06 seconds with BVH acceleration



banana.dae renderred in 21 seconds with normal shading WITHOUT BVH acceleration

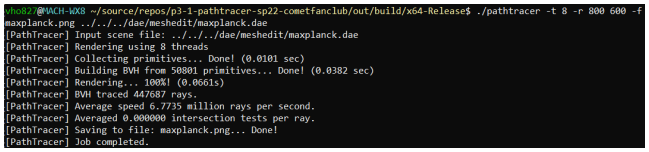
cow.dae renderred in 34 seconds with normal shading WITHOUT BVH acceleration

the progress made in 2 minutes to render maxplanck.dae with normal shading WITHOUT BVH acceleration



banana.dae renderred in 0.14 seconds with normal shading WITH BVH acceleration

cow.dae renderred in 0.04 seconds with normal shading WITH BVH acceleration



maxplanck.dae renderred in 0.06 seconds with normal shading WITH BVH acceleration

CBlucy.dae renderred in 0.05 seconds with normal shading WITH BVH acceleration

Part 3: Direct Illumination:

Direct Lighting Implementation Walkthrough

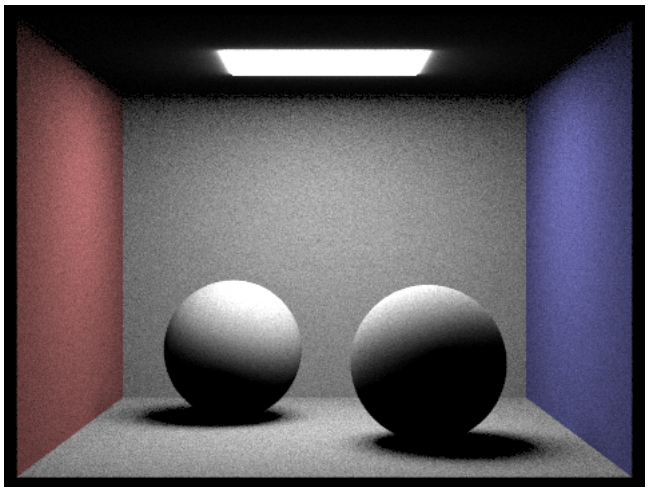
Uniform Hemisphere Sampling

To estimate the direct lighting at a given point, the uniform hemisphere sampling method involves taking a specified number of uniformly random sample directions within the hemisphere about that point. Then for each sample direction, trace a ray out in that direction (in the world frame) and determine if the ray intersects any other primitives in the scene within the lifespan of the ray. If there isn't an intersection, then this direction contributes nothing to the direct lighting at the given point. If there is an intersection, however, then we take into consideration the irradiance coming from the object the ray intersected. These irradiance values are included in one of the terms in the Monte Carlo estimation of the reflectance equation, which eventually represents the amount of outgoing light.

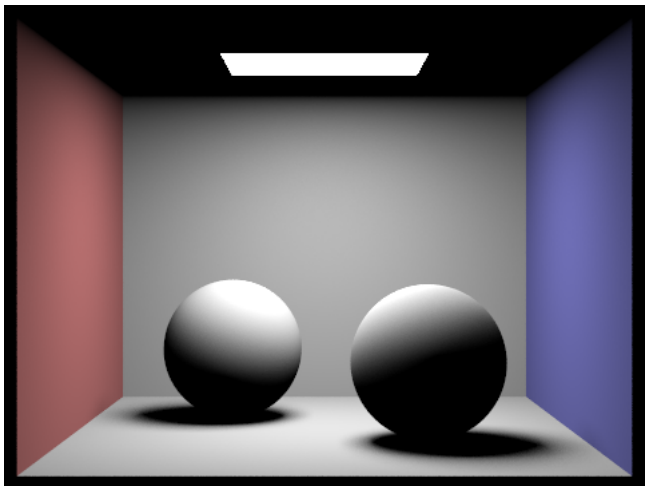
Importance Sampling Lights

To estimate the direct lighting at a given point, the importance sampling of lights operates differently from the Uniform Hemisphere Sampling method. Rather than taking uniform random samples in a hemisphere, this method iterate thorugh every light source in the scene. Depending on the type of light source, the method will either trace a ray (in the world frame) to that light source (point light source) or will trace rays to multiple points on the light sampled randomly (area light source). For each of these rays, if there is an intersection before the ray reaches the light source, then this sample on the light source contributes nothing to the direct lighting at the given point. If there is NOT an intersection with another primitive before the ray reaches the light source, then we take into consideration the light emitted by the light source at the point we sampled and include it in the total outgoing light from our original given poitn. This is somewhat opposite from the uniform hemisphere sampling method in which illuminance from a direction contributes to the outgoing light if there was an intersection in that direction. For area lights, the contributions of the rays for all samples on that area light are averaged to find the average contribution. The outgoing light from a given point due to all light sources are accumulated to equal the outgoing light from the given point

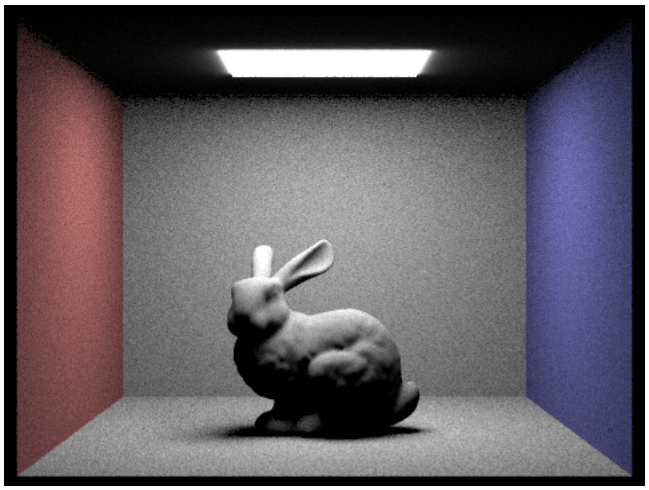
Some Images Renderred with Both Implementations of Direct Lighting Function



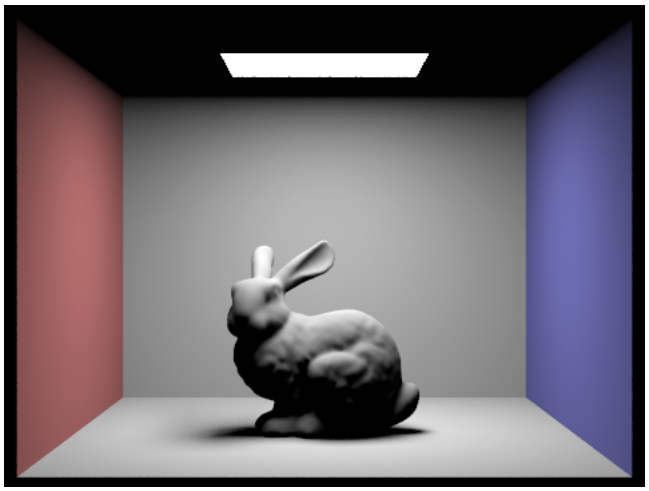
CBSpheres_lambertian.dae renderred with the uniform hemisphere sampling implementation of the direct lighting function



CBSpheres_lambertian.dae renderred with the importance sampling lights implementation of the direct lighting function, demonstrating dramatically reduced noise in the image

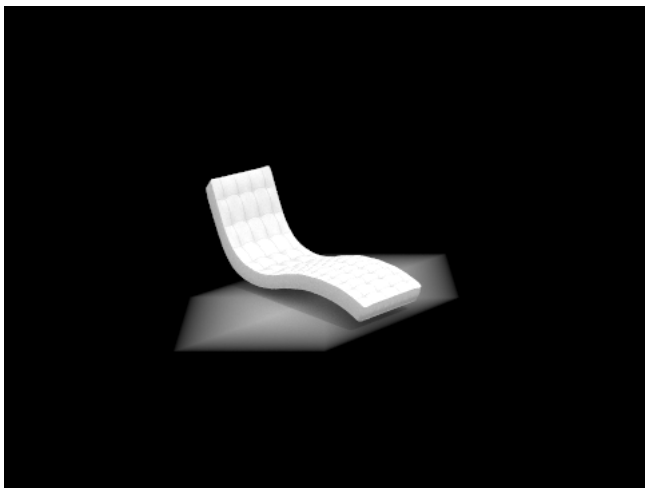


CBbunny.dae renderred with the uniform hemisphere sampling implementation of the direct lighting function



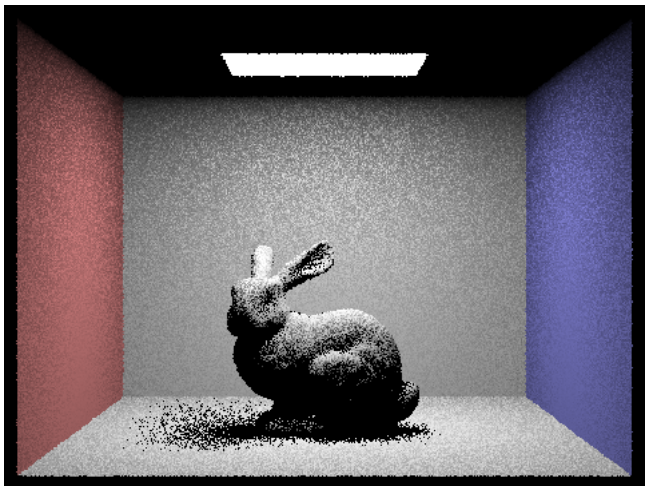
CBbunny.dae renderred with the importance sampling lights implementation of the direct lighting function, demonstrating dramatically reduced noise in the image

bench.dae renderred with the uniform hemisphere sampling implementation of the direct lighting function. This scene is black because there aren't other primitives in the scene to act as light sources and it's unlikely that the randomly sampled rays directions are pointing in the right direction to reach the light source

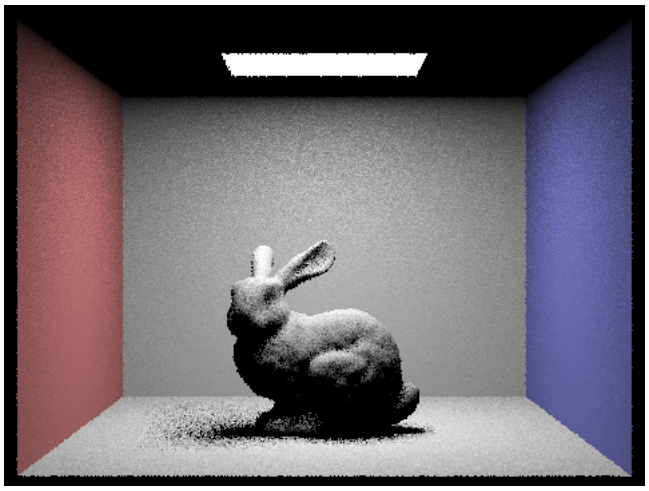


bench.dae renderred with the importance sampling lights implementation of the direct lighting function. The bench actually appears in this render becuase we explicitly traced rays to all light sources in the scene

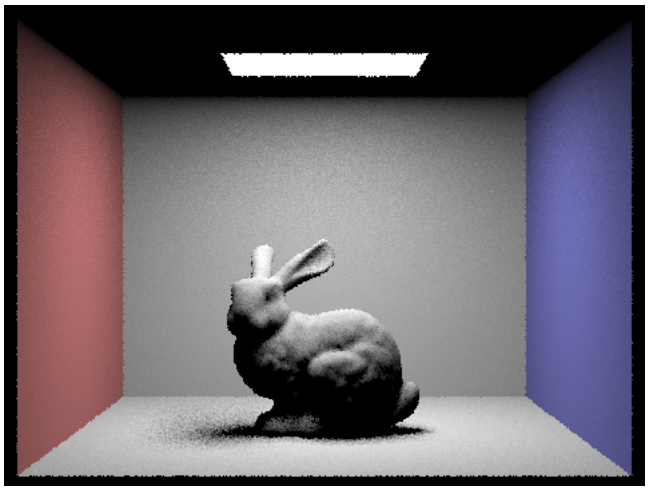
Comparison of Noise Levels in Soft Shadows in CBbunny.dae with varying numbers of light rays, when renderred with 1 sample per pixel and the importance sampling implementation of the direct lighting function



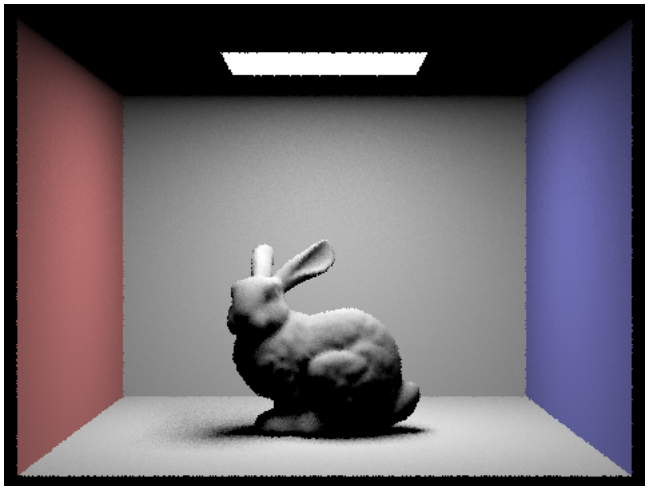
CBbunny.dae renderred with 1 light ray, showing a significant amount of noise in soft shadows. The edges of the shadow of the bunny as well as in the corners of the room don't fade smoothly, instead are seen as wide pools of heavy shadow.



CBbunny.dae renderred with 4 light ray, shows significant improvement in the noise in the soft shadows compared to when renderred with 1 light ray



CBbunny.dae renderred with 16 light ray shows lower levels of noise in the edges of shadows shows significantly reduced intensity and much better diffused as seen from the shadows' edges being more grey than black now



CBbunny.dae renderred with 64 light ray, showing significantly less noise in soft shadows, as witnessed in the edges of the bunny's shadow and the corners of the room, the shadow diminishes gracefully and smoothly in soft shadow regions.

Comparison of importance and uniform hemisphere light sampling

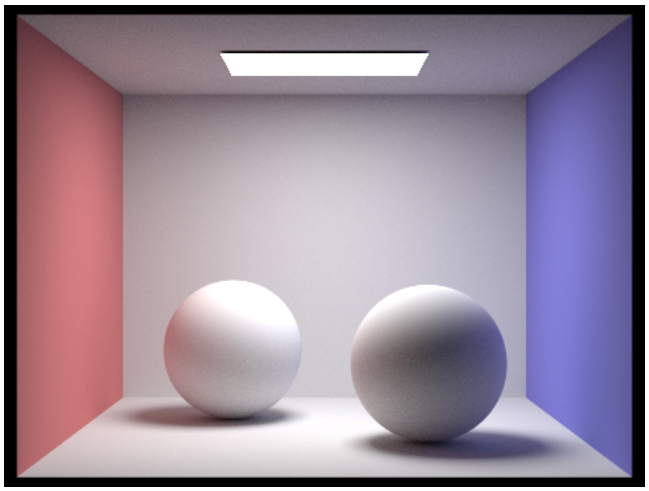
As we can see above, the results differ drastically for direct lighting by importance sampling and for direct lighting with uniform hemisphere sampling. The first major difference is that for our point light source, the bench, the bench is not rendered at all for the hemisphere sampling. This is because it is unlikely that the exact direction towards the point light is sampled with this mode, so our algorithm simply doesn't find any sources of illuminance in uniformly sampling across the hemisphere for some point/pixel. In comparison, in importance sampling we only sample the valid directions from the bench to the point light source, allowing us to actually detect the point light source. For the other two images, the spheres and the bunny, we can see that the importance sampling yields a much less noisy image. While the uniform hemisphere sampling results in a lot of graininess due to a high variance in samples across the hemisphere and missing the light source in the vast majority of the samples, our importance sampled images are very smooth because the samples are dedicated towards directions that actually contribute or are likely to contribute to our lighting.

Part 4: Indirect Illumination:

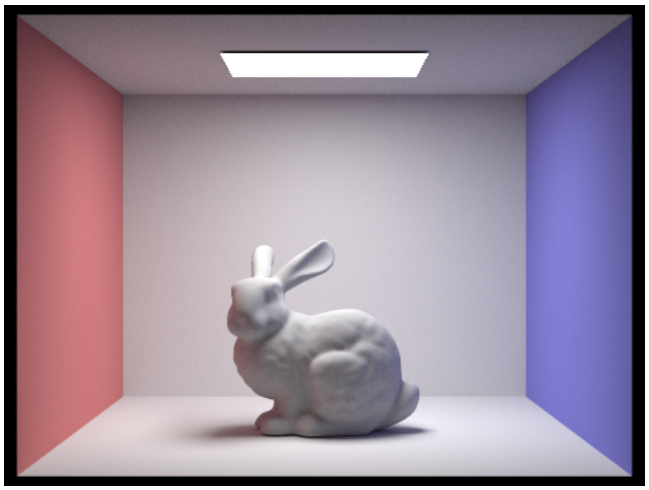
Walk through of our implementation for the indirect lighting function.

Our implementation of the indirect lighting function starts off with the one bounce direct lighting. Then it determines whether or not it should be considering any subsequent bounces and add to the indirect lighting term. Given a sample direction, the criteria used to determine whether or not additional bounces should be considered can be broken down into 3 parts. First, if the randomly sampled direction does NOT intersect with any other primitives in the scene, then we can't consider any subsequent bounces. Second, if the Russian roulette probability coinflip does NOT result in a continue, then we won'tconsider any subsequent bounces. Finally, if the max_depth number of bounces has been reached, then we won't consider any subseuent bounces either, where a max depth of 0 means no SUBSEQUENT bounces and only includes direct lighting. If all three criteria still determines that we should consider subsequent bounces to add to the indirect lighting term, we just apply the indirect lighting function process recursively but from the new intersection location in the sample direction.

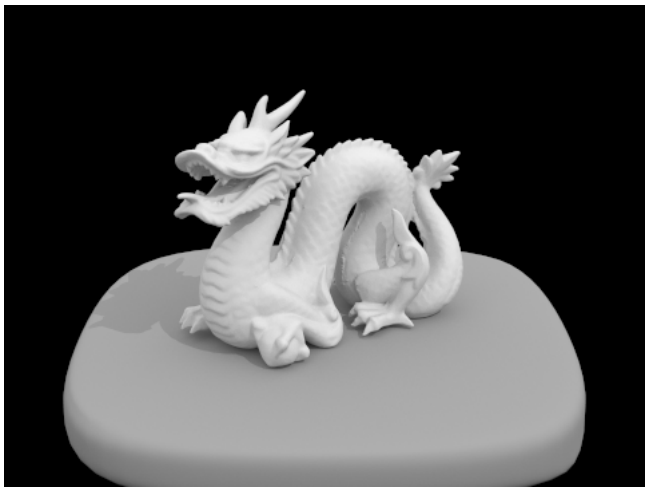
Some Images Rendered with Global (direct and indirect) Illumination at 1024 Samples Per Pixel



CBSpheres_lambertian.dae renderred with global illumination at 1024 samples per pixel.

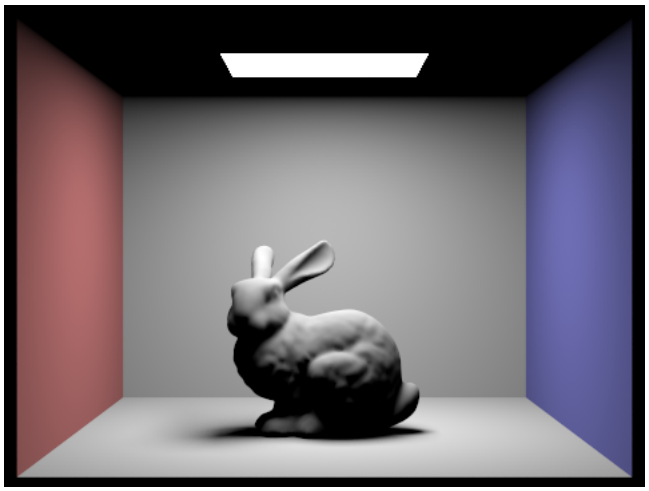


CBbunny.dae renderred with global illumination at 1024 samples per pixel.

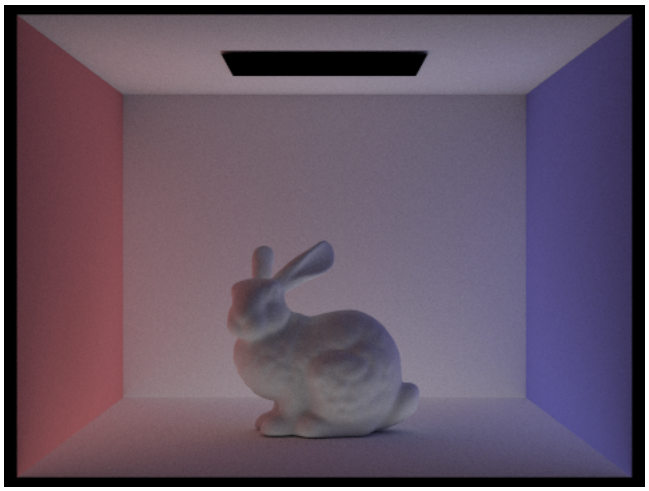


dragon.dae renderred with global illumination at 1024 samples per pixel.

Comparison of One Scene Renderred with Only Direct Illumination, then Only Indirect Illumination at 1024 Samples Per Pixel



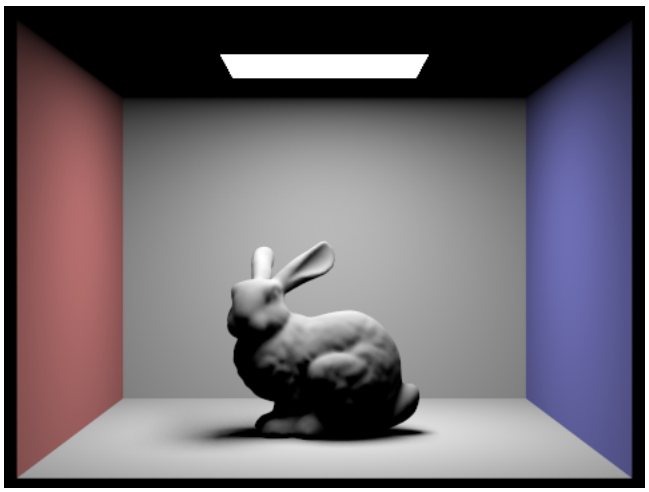
CBbunny.dae renderred with only direct illumination at 1024 samples per pixel.



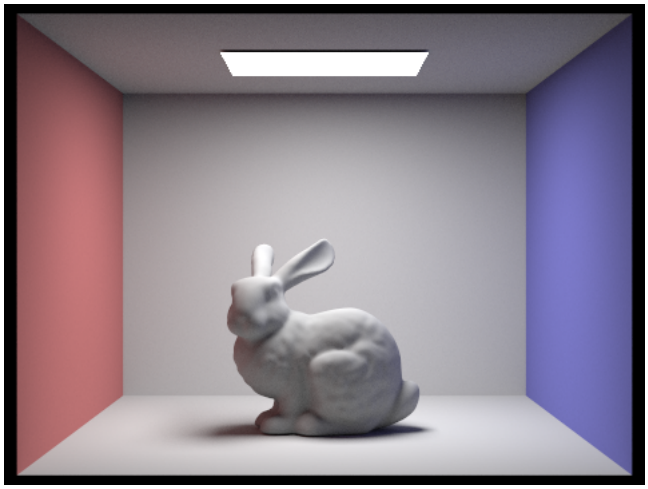
CBbunny.dae renderred with only indirect illumination at 1024 samples per pixel.

As we can see, the direct lighting yields an image closer to the image with both direct and indirect illumination, with very hard, black shadows. The indirect illumination image is brighter where the dark shadows in the indirect illumination are, i.e. on the ceiling and in the shadows of the bunny, while the areas hit by harsh direct lighting (the back of the bunny) are darker as there's less indirect lighting there.

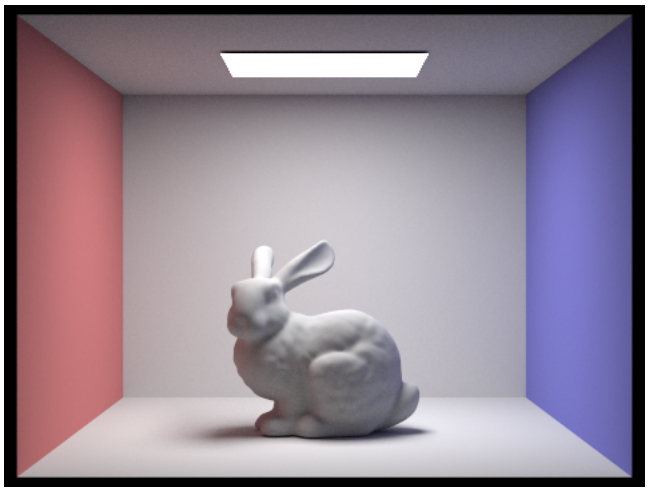
For CBbunny.dae, compare rendered views with max_ray_depth set to 0, 1, 2, 3, and 100 (the -m flag). Use 1024 samples per pixel.



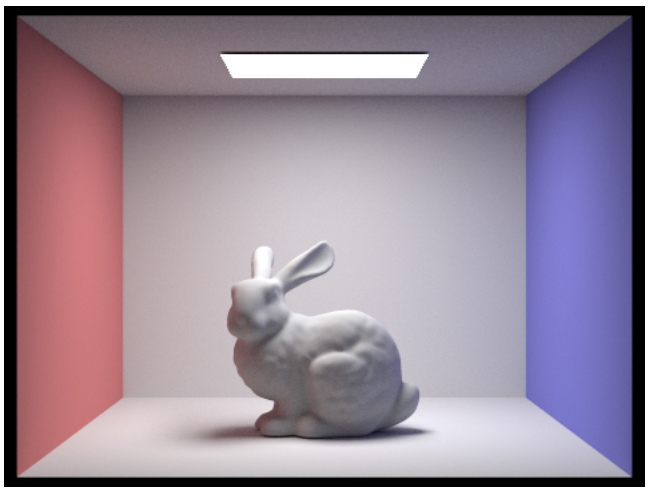
CBbunny.dae renderred with max_ray_depth set to 0 at 1024 samples per pixel, which is basically just one bounce (direct) illumination.



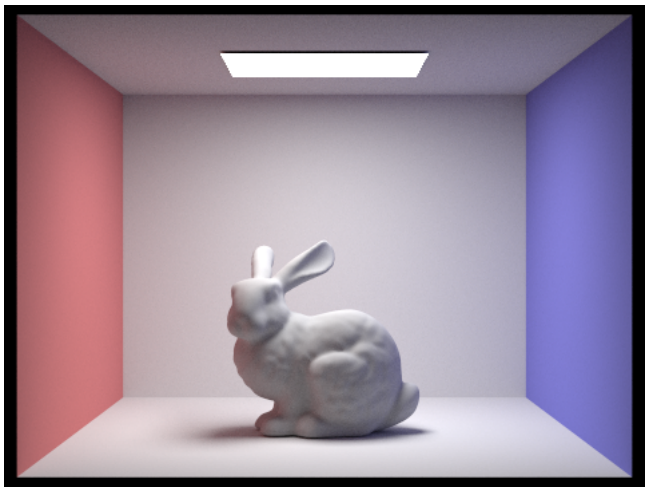
CBbunny.dae renderred with max_ray_depth set to 1 at 1024 samples per pixel.



CBbunny.dae renderred with max_ray_depth set to 2 at 1024 samples per pixel.



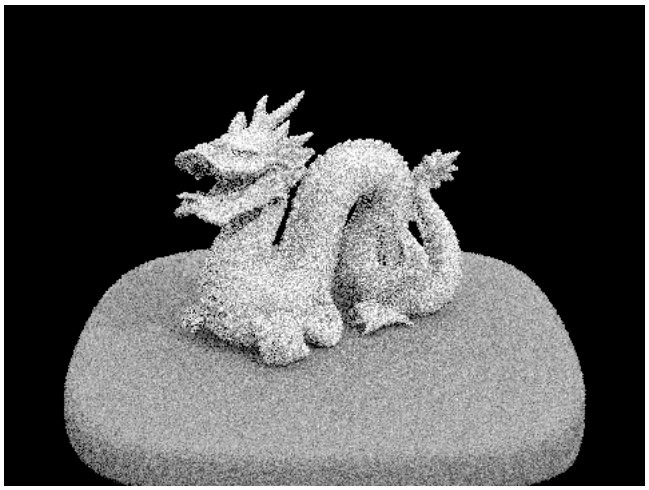
CBbunny.dae renderred with max_ray_depth set to 3 at 1024 samples per pixel.



CBbunny.dae renderred with max_ray_depth set to 100 at 1024 samples per pixel.

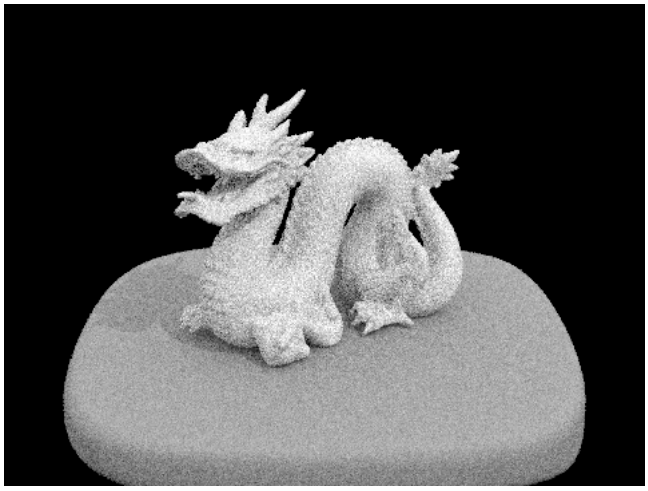
Here we can see that the max ray depth of 0 produces the direct lighting illumination image, and the subsequent images have softer and softer shadows with diminishing diffrences as ray depth increases (3 and 100 are very similar)

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.

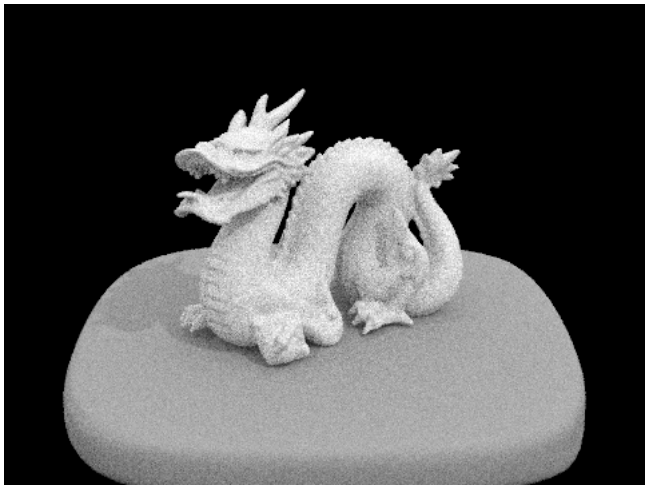


dragon.dae rendered with 1 sample per pixel and 4 light rays.

dragon.dae rendered with 2 sample per pixel and 4 light rays.

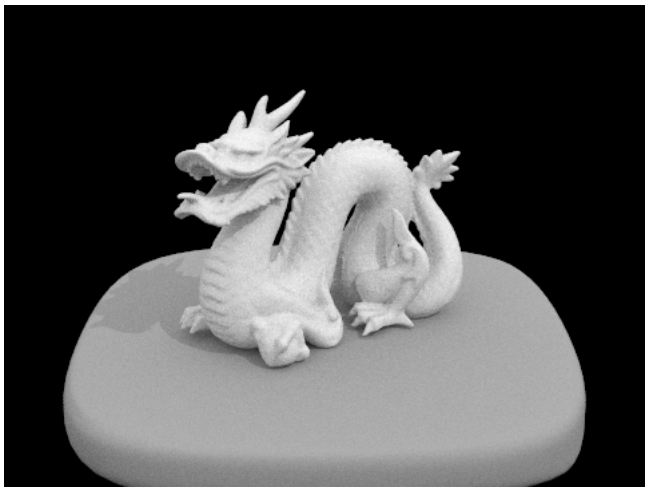


dragon.dae rendered with 4 sample per pixel and 4 light rays.

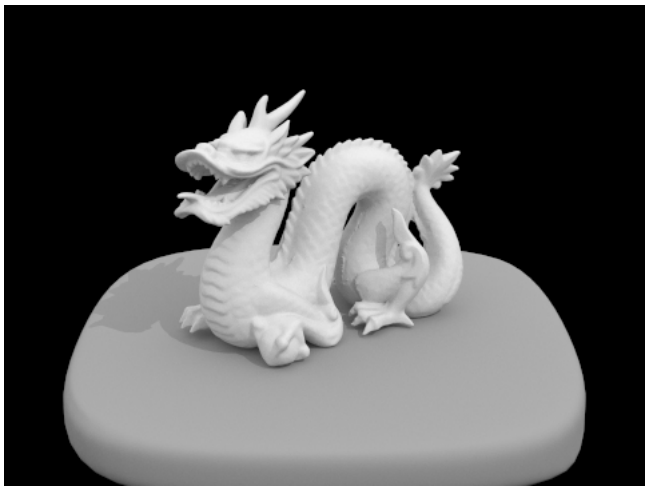


dragon.dae rendered with 8 sample per pixel and 4 light rays.

dragon.dae rendered with 16 sample per pixel and 4 light rays.



dragon.dae rendered with 64 sample per pixel and 4 light rays.



dragon.dae rendered with 1024 sample per pixel and 4 light rays.

Here, we can see that our first images with few samples per pixel are extremely noisy, as you'd expect, and as the samples per pixels increases the variance in the lighting of each pixel decreases, and visual noise decreases significantly.

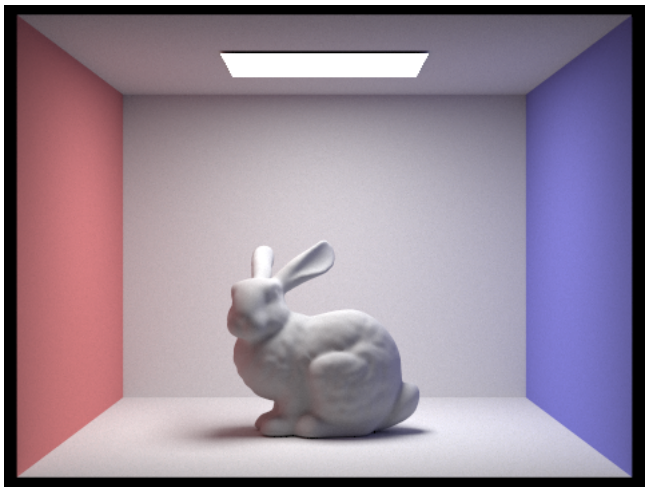
Part 5: Adaptive Sampling:

Adaptive Sampling Implementation

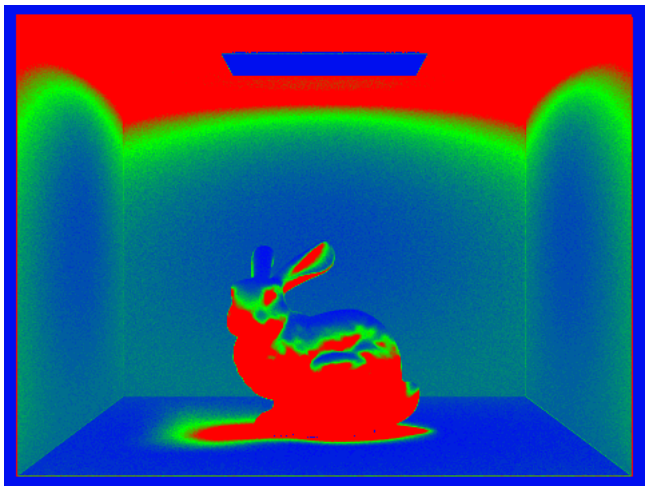
To implement adaptive sampling, we used the algorithm described in the spec. Instead of the previous implementation of a fixed n samples per pixel, we terminate sampling when a function of the variance of the samples, number of samples, and mean of the samples has been fulfilled. Once our samples have converged to the desired degree, we stop sampling.

We define two more variables, s1 and s2, to help us check after each sample if we have converged enough to stop sampling. We store in s1 a running sum of all of the illuminance values of the samples, and we store in s2 a running sum of the square of the illuminance values of the samples. From these two values, we can calculate our mean and variance of our samples so far: the mean is defined as s1 divided by the number of samples, and the variance is defined as sqrt((1/(n-1)) * (s2 - s1**2/n)), where n is our number of samples. From these statistics, we can calculate our heuristic value I, which is defined as 1.96 * variance / sqrt(numsamples). We want to terminate sampling when I <= maxtolerance * mean, where maxtolerance is a hyperparamter (0.05 by default.) For efficiency, we only calculate the heuristic once every numsamplesperbatch samples, where numsamplesberpatch is a hyperparameter (default 32), and terminate sampling once this check passes or we reach the maximum allowed number of samples.

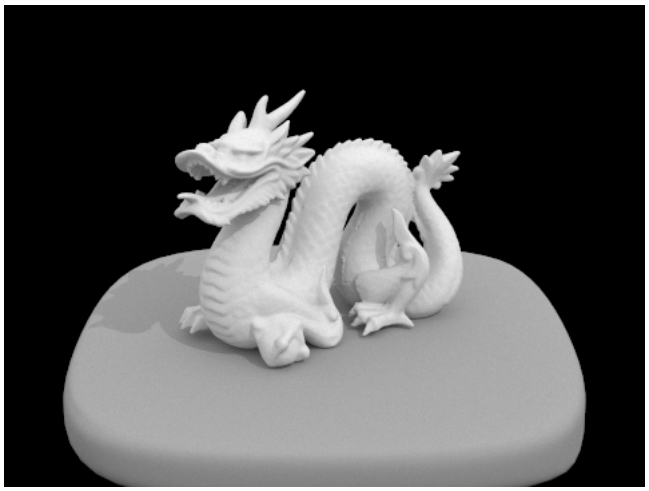
Example images showing the image rendered using adaptive sampling as well as the sample rate image, where red signifies the highest sampling rate and blue signifies the lowest sampling rate.



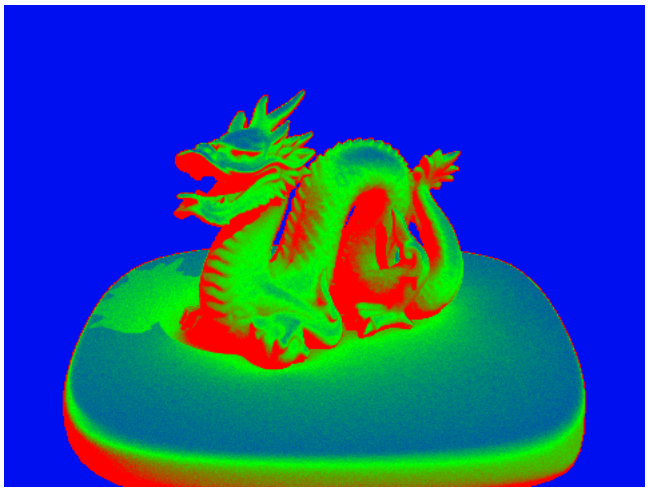
bunny rendered with adaptive sampling



sampling rates of bunny sampled with adaptive sampling



dragon rendered with adaptive sampling<



sampling rates of dragon sampled with adaptive sampling<

We see here that the shadows and darker areas have significantly higher sampling rates than those in direct light. This is because for these areas, the variance in the samples was much higher. This is also why these areas were much noisier without adaptive sampling. Thus, using this heuristic, we naturally sample noisier areas more