CS 184: Computer Graphics and Imaging, Spring 2023

Project 3-1: Path Tracer

I-Lun Tsai, Hsuan-Hao Wang

Website URL

Overview

Path tracing is a method that simulates the behavior of light rays. In this project, we implemented a renderer that uses a path tracing algorithm to render scenes with physical lighting. Specifically, I generated random rays from every pixel of the camera, determined the nearest intersection point for the ray with a primitive in the scene.

After determining the intersection points, we rendered lighting calculating the amount of light that reached each pixel of the camera which included direct lighting and global illumination. This step was crucial in creating a more realistic illumination in 3D scenes.

In summary, we were able to simulate the behavior of light rays in a scene, generate realistic illumination in 3D scenes, and speed up the rendering process using optimization techniques such as bounding volume hierarchy.

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

Ray Generation

Simulating light involves casting a ray from its origin coordinate, which is the camera's position in the world, toward its direction vector. The process of casting a ray from a pixel contains three coordinate transformations to determine the appropriate ray direction.

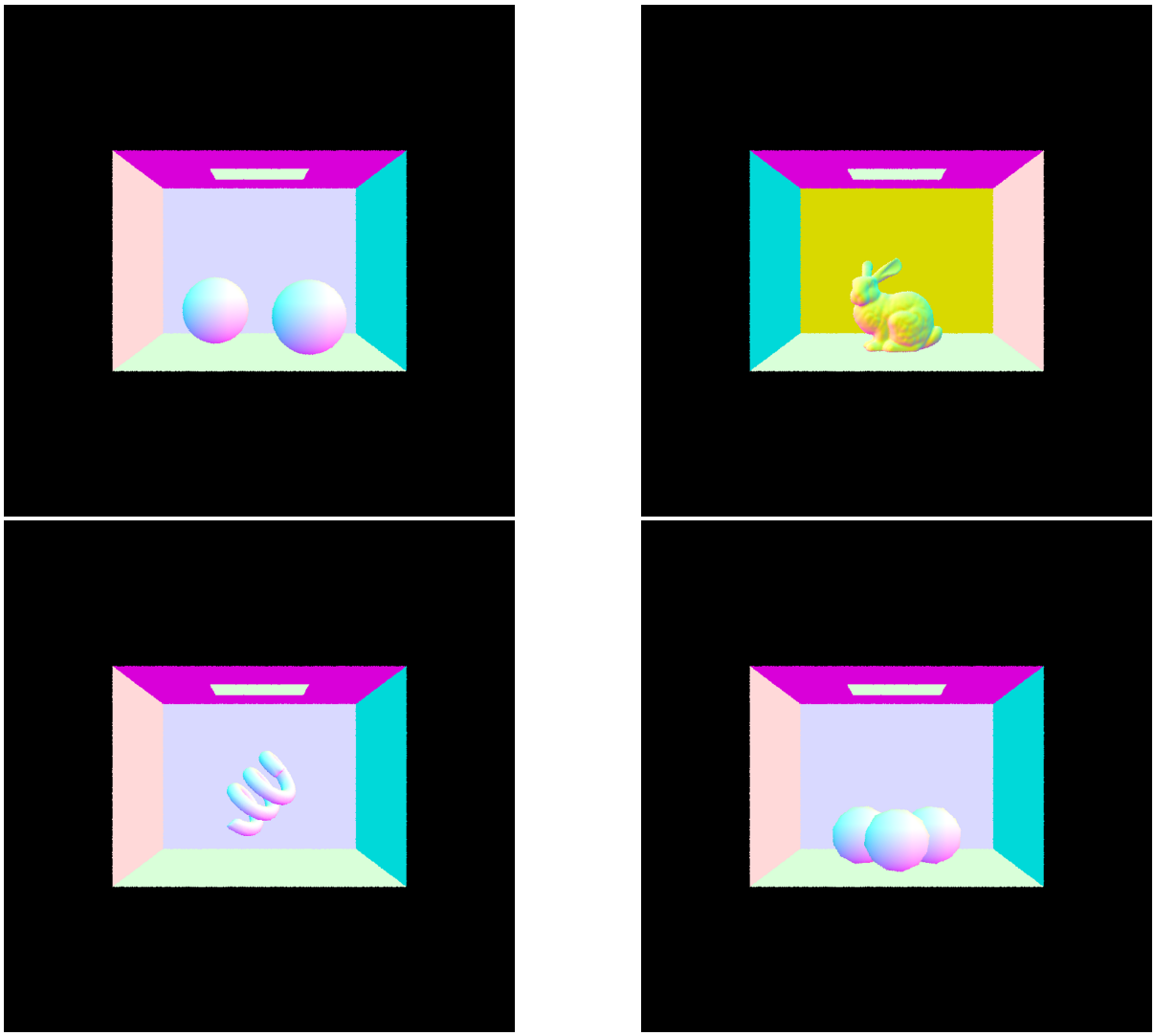
1. Pixel coordinates frame $\rightarrow$ normalized coordinate frame (scaling axes)
2. Normalized coordinate frame $\rightarrow$ image frame (translation and scaling)
3. Image frame $\rightarrow$ world frame (camera-to-world transform matrix)

We can sample a single pixel multiple times by altering the ray direction. After determining the ray direction for the pixel, the direction vector undergoes a camera-to-world transformation, ensuring that the ray propagates correctly in the scene.

Primitive Intersection

We used the Moller-Trumbore Algorithm to detect ray intersections with a triangle primitive without the plane equation on which the triangle is laying. This algorithm utilizes the three vertices of the triangle to find a vector normal to the plane which is further used to calculate the ray-plane intersection by solving linear equations. Barycentric coordinates are then used to determine whether the intersection point resides inside a triangle primitive. We had to check whether the t-value of the intersection was within the maximum and minimum bounds of the scene.

For ray-sphere intersections, the intersection t-values were computed by solving a quadratic equation using the quadratic root formula. We checked if either of the roots of the quadratic fell within the maximum and minimum bounds and selected the minimum one.



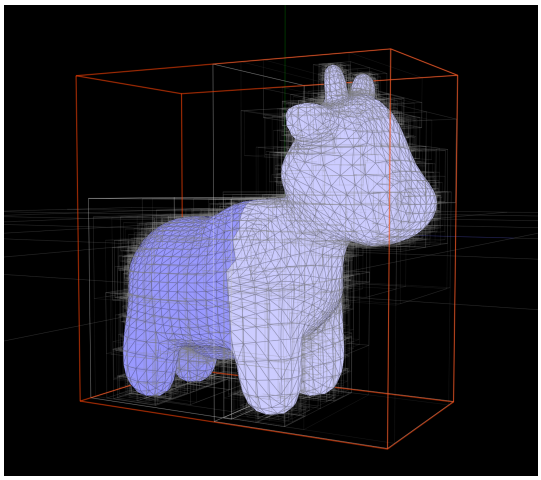
Part 2: Bounding Volume Hierarchy (20 Points)

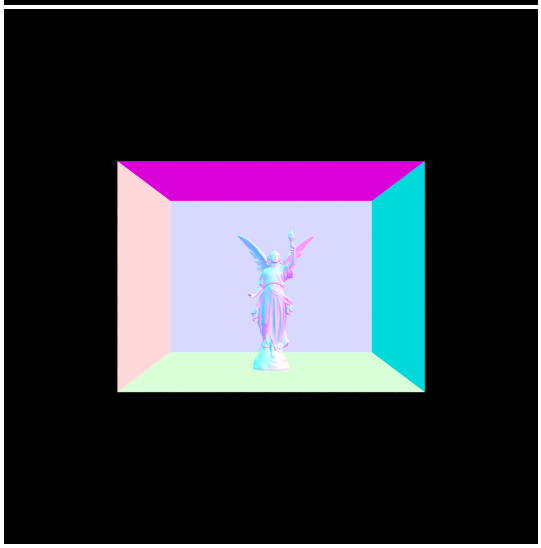
Our algorithm in constructing a Bounding Volume Hierarchy (BVH) is as follows:

1. Computed the mean 3D position of the centroids of each primitive within the node's bounding box. The (x, y, z) coordinates are the 3 candidate planes along which to split.
2. Find which split would result in the minimum (surface area for node's bounding box $\times$ number of primitives) for both left and right child nodes.
3. Recursively construct the BVH for the left and right nodes until leaf node contains at most max_leaf_size primitives.

We reason that selecting the minimum surface area from the 3 axes rather than a random axis along which to split would result in bounding box filled densely with primitives. Therefore, the probability that a ray would hit the bounding box of a BVH and also hit a primitive was largely increased.

The computational efficiency is significantly increased by BVH. Theoretically, the average number of intersection tests per ray is O(n) without BVH and O(logn) with a BVH, where n is the number of primitives in the scene. In order to render the image of a cow below, approximately 872 ray intersection tests were performed without the BVH compared to only 4 intersection tests with the BVH. In terms of rendering time, the process took 41.9231 seconds to complete without the BVH compared to 0.0362 seconds with the BVH.

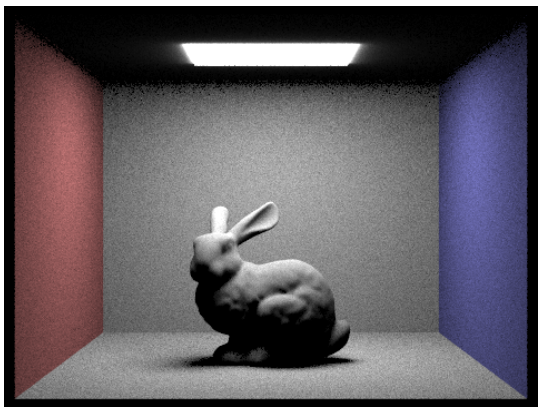


Part 3: Direct Illumination (20 Points)

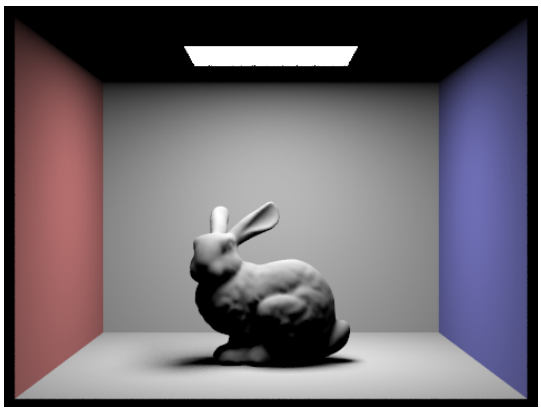
Two ways to achieve direct lighting are uniform hemisphere sampling and light importance sampling. Uniform hemisphere sampling utilizes a probability distribution function (pdf) that evenly distributes the probability among all points in the hemisphere. To simulate a bounce from a light source to a surface and back to the camera, we randomly generate a vector from this hemisphere and then check if the resulting ray intersects any light sources. On the other hand, light importance sampling involves sampling over the light sources and checking if there is an unobstructed path for a ray to travel between the light and the surface. By utilizing the rays and light values, we can calculate the radiance produced on the surface by each light source and then average over all the light sources.

Uniform Hemisphere Sampling

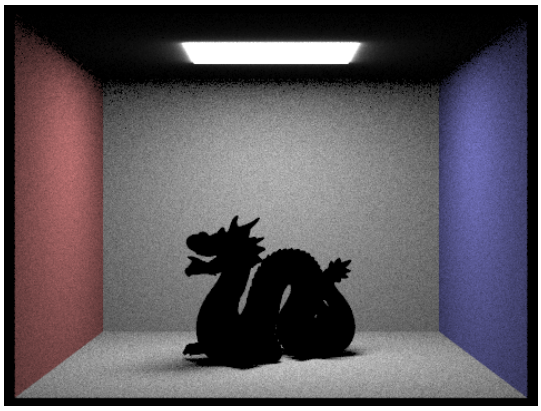


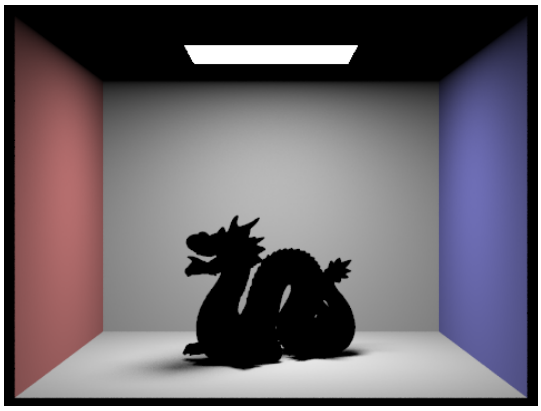
CBbunny.dae

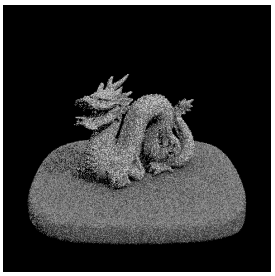
Light Sampling

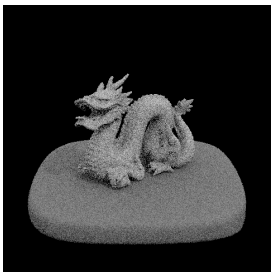


CBbunny.dae


CBdragon.dae


CBdragon.dae


1 Light Ray


4 Light Rays


16 Light Rays

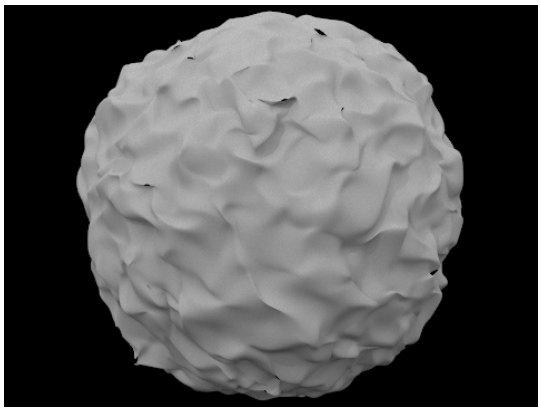

64 Light Rays

Since we take multiple samples for non-point light sources, the randomness arises from the location differences along the light source. As the number of samples increases, the sensitivity of the surface reflectance to the variance in the distance from the sampled light source decreases. With lower sampling rates, the chances of accidentally sampling vectors that are not representative of the lighting of the point increase. For example, we might end up sampling a shaded point when most of the light reaches the point. However, as we increase the number of samples, we account for all of these vectors more effectively, which results in a more noticeable shade gradient.

Uniform hemisphere sampling yields a higher noise level, giving equal weight to samples from all locations across the hemisphere. Uniform hemisphere sampling is impractical for environments with point lights because the sampled rays will not point directly to the light. Compared to light importance sampling, the environment tends to have more noisy light points. Light importance sampling produces a more consistent image with an observable level of consistency. For example, in a single-bounce light-sampled image, the ceiling is always completely dark, as the rays between the surface and the light are always regarded as being shadowed. In general, light importance sampling results in a smoother and more consistent image with the geometries, but it has a somewhat artificial look that can be overproduced.

Part 4: Global Illumination (20 Points)

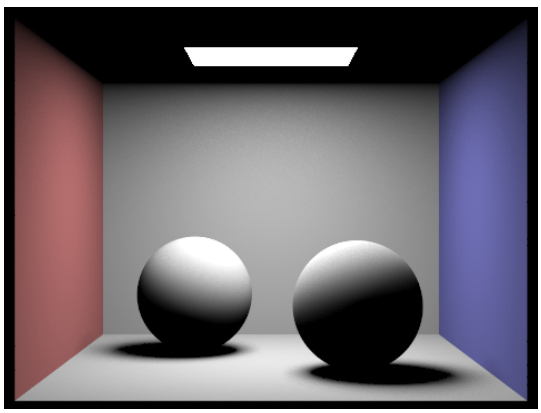
Although direct lighting produces decent images, it is unrealistic because it only considers one light bounce. In real life, light bounces multiple times off surfaces, so areas not directly hit by light can still be illuminated. Therefore, global illumination is needed to account for this.



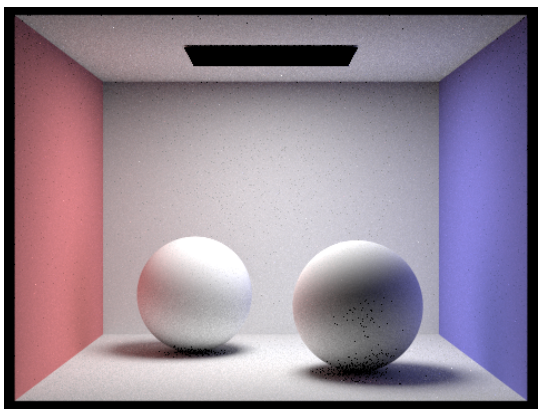
In this section, the first step in implementing global illumination was to create a method that returns the radiance at zero light bounces. This method determines the light emitted from a light source that travels directly toward the camera. This method uses direct lighting methods created in Part 3, with hemisphere sampling or importance sampling.

The recursive method for at least one light bounce was then implemented. This method uses Russian Roulette to terminate sampling randomly to decrease the rendering computations required for the scene. The method starts by setting a vector L_out, which is used to accumulate radiances. The maximum ray depth is then checked to ensure it is greater than one. If the pdf is zero, dividing by it can lead to Unsavory White Specks. Also, if the new ray shot from the current point does not intersect with the BVH, continuing is useless. If these conditions are met, the method checks if the current ray depth is the same as the maximum. If it is, another bounce must be guaranteed, so the method accumulates L_out with a recursive call to itself. The resulting radiance is scaled by the BSDF and a cosine factor, then divided by the pdf. In this recursive call, the new ray created earlier is used, with a depth value decremented from the current ray's value. This approach eventually terminates the recursion without Russian Roulette because the ray depth will no longer exceed one.

If the current ray is less than the maximum ray depth, Russian Roulette can be used to terminate the ray randomly. A coin_flip method is called with a termination probability of 0.3 (the same as the continuation probability of 0.7). If Russian Roulette dictates that sampling should continue, the method checks if the current ray's depth is greater than one. If it is, L_out is accumulated with a recursive call to itself, using the new ray created earlier. As before, the resulting radiance is scaled by the BSDF and a cosine factor, then divided by the pdf. However, it is also divided by (1 - termination probability) because of the use of Russian Roulette.



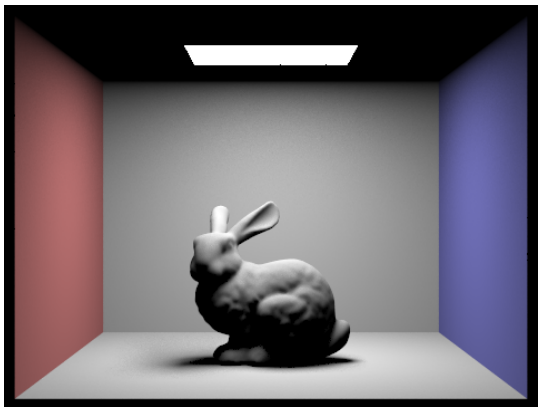
Only direct illumination

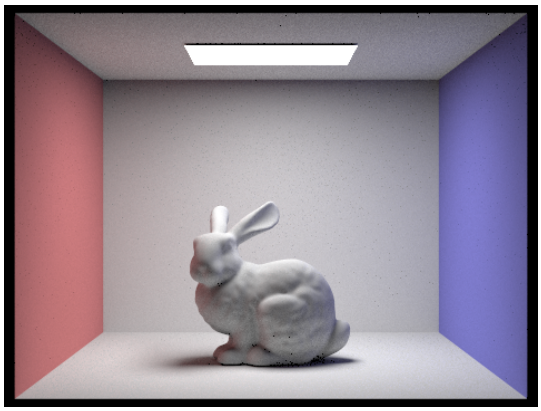


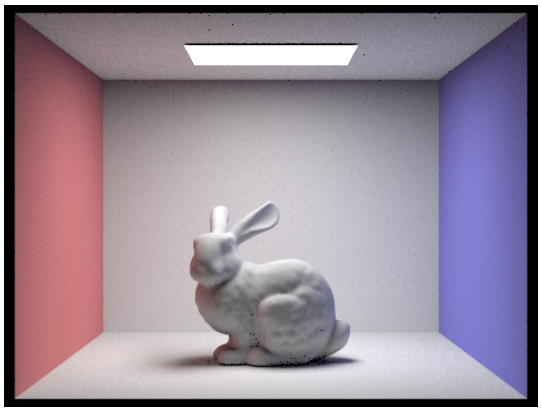
Only indirect illumination

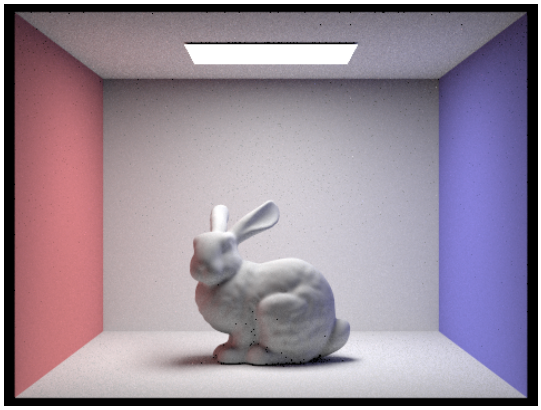
The images of CBbunny.dae displayed below reveals that increasing the maximum ray depth leads to an increase in the scene's light. Initially, with zero bounces, the only visible light is emitted from the light source. After one bounce, the resulting image is almost identical to that obtained in Part 3, where direct lighting and importance sampling was used, except for the visible emitted light at the top. As the maximum ray depth is increased, the resulting image becomes progressively brighter, and more colored light is seen on uncolored surfaces. It is important to note that, despite the probabilistic random termination used in our implementation, the render time for images at higher maximum ray depths did not increase significantly compared to images at lower maximum ray depths. Even with a maximum ray depth of 100, the likelihood of the at_least_one_bounce method being recursed 99 times is extremely low (in our implementation, the probability of this happening is 0.7^{99}, which is a very small number).


max_ray_depth = 0 (CBbunny.dae)

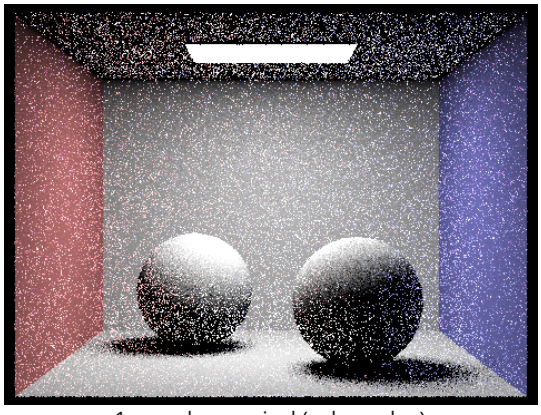

max_ray_depth = 1 (CBbunny.dae)

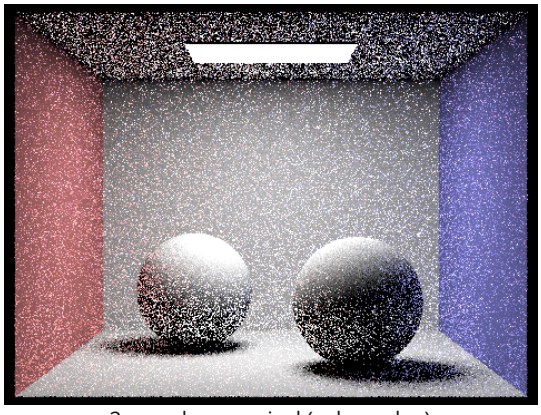

max_ray_depth = 2 (CBbunny.dae)

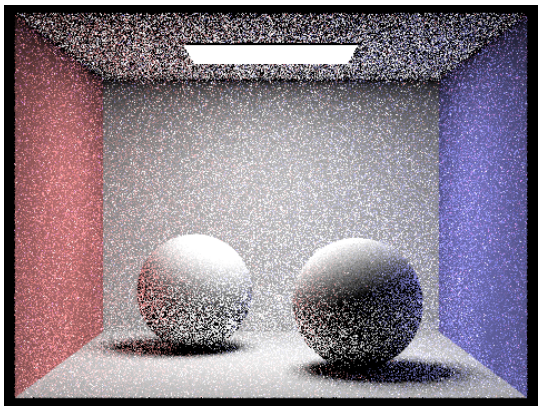

max_ray_depth = 3 (CBbunny.dae)

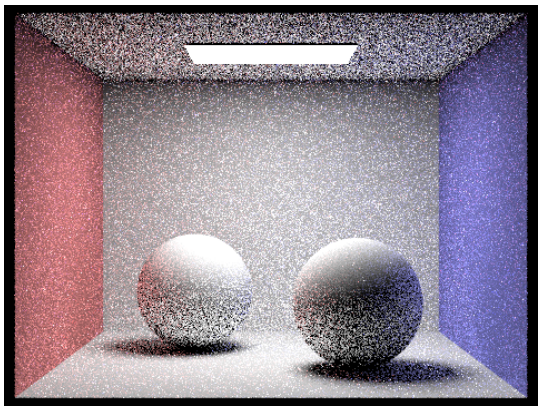

max_ray_depth = 100 (CBbunny.dae)

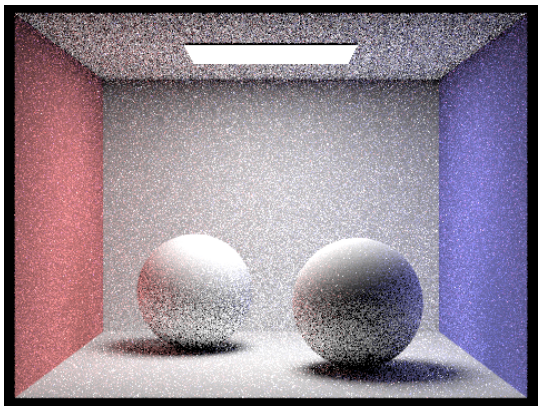
We can see that increasing sample size decreases the noise of the render from the below comparisons.

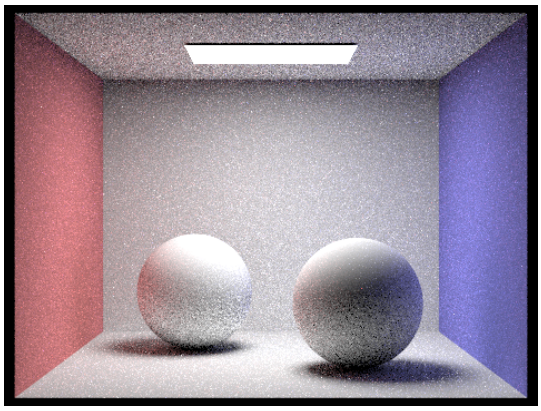

1 sample per pixel (sphere.dae)

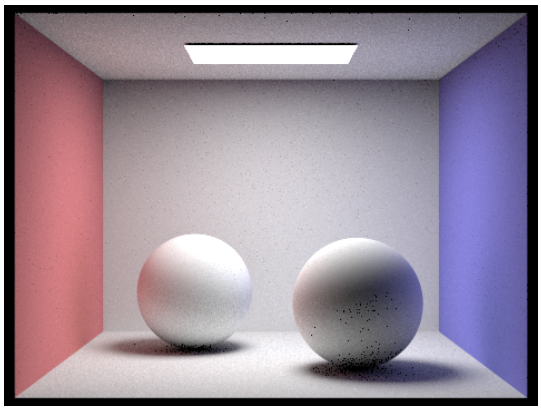

2 samples per pixel (sphere.dae)


4 samples per pixel (sphere.dae)


8 samples per pixel (sphere.dae)

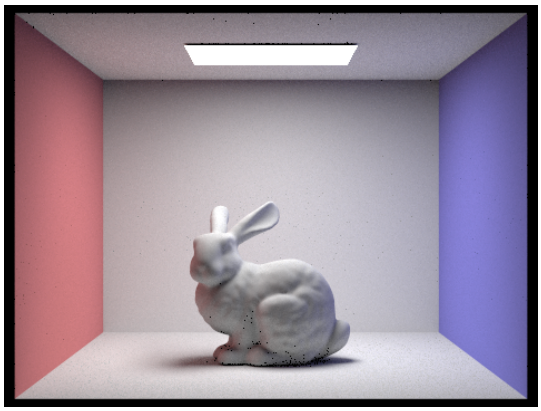

16 samples per pixel (sphere.dae)


64 samples per pixel (sphere.dae)

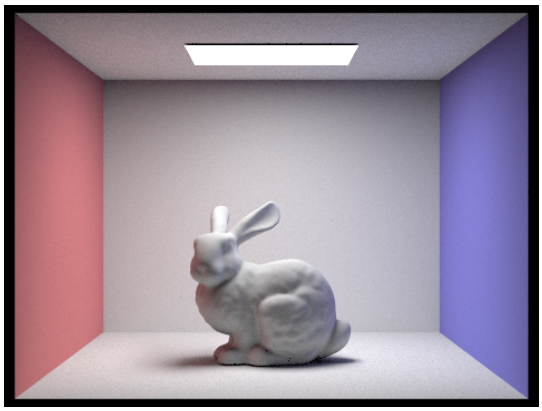

1024 samples per pixel (sphere.dae)

Part 5: Adaptive Sampling (20 Points)

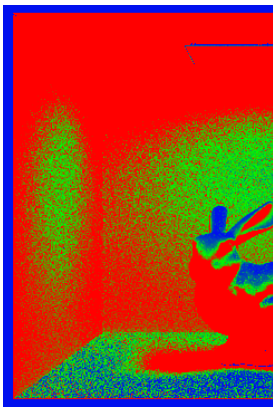
Adaptive sampling is a method that intelligently stops processing rays per pixel by analyzing the convergence behavior of the pixel's illuminance. If a pixel appears to have converged to a stable value, then no additional sampling is necessary to render the scene accurately. The implementation of this technique involves adjusting the raytrace_pixel() function so that it ends its sample-averaging loop before reaching the maximum number of samples, provided that the conditions for illuminance convergence have been met. These conditions are based on the average illuminance, the variance in the samples, and z-scores, assuming that the probability distribution of illuminance in the pixel's footprint is Gaussian. This technique improves computational efficiency by optimizing the amount of pixel super-sampling, and it is similar to Mipmap architectures because it combines the information variation within a single pixel's scene footprint with the degree of super-sampling required for that pixel.



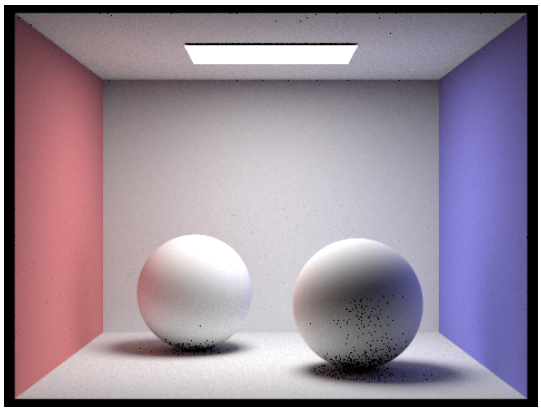
Without adaptive sampling (CBbunny.dae)



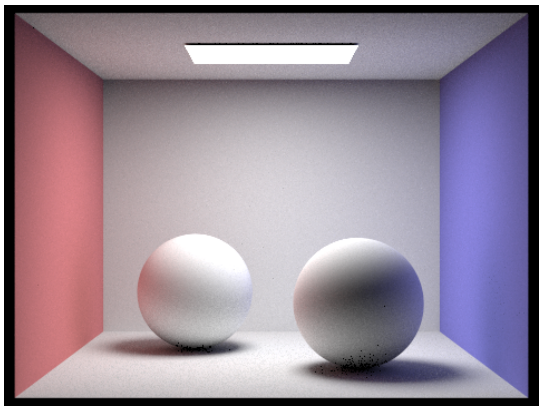
With adaptive sampling (CBbunny.dae)



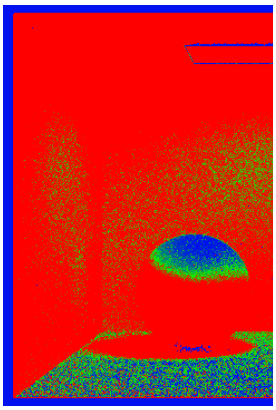
Sample rate imag



Without adaptive sampling (CBspheres_lambertian.dae)



With adaptive sampling (CBspheres_lambertian.dae)



Sample rate image (CBs