Assignment 3.1: PathTracer

Nico Deshler

In this project, I developed a radiometrically accurate ray-tracing pipeline for rendering virtual scenes. The pipeline is composed of five parts, each of which contain specific optimizations to improve render efficiency and quality. Overall, these parts, and their effects on the render, are presented in the chronology in which they are executed within the pipeline. The images shown herein attempt to visually demonstrate the role of each step of the render. The differences in render quality and efficiency achieved through the progression of algorithms explored in this project are salient.

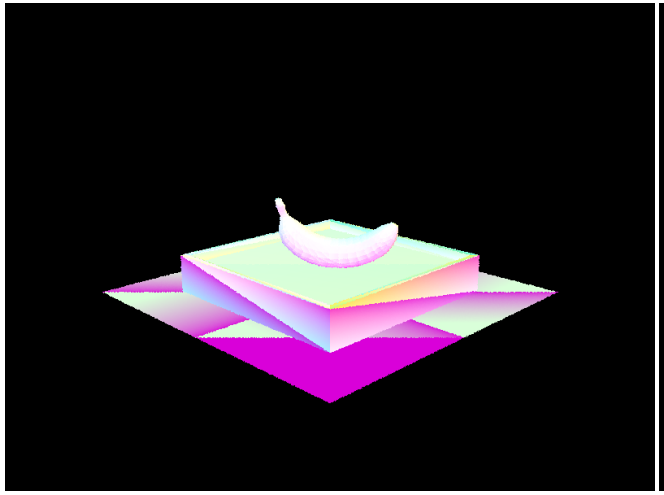
Part 1: Ray Generation and Scene Intersection

Ray Generation:

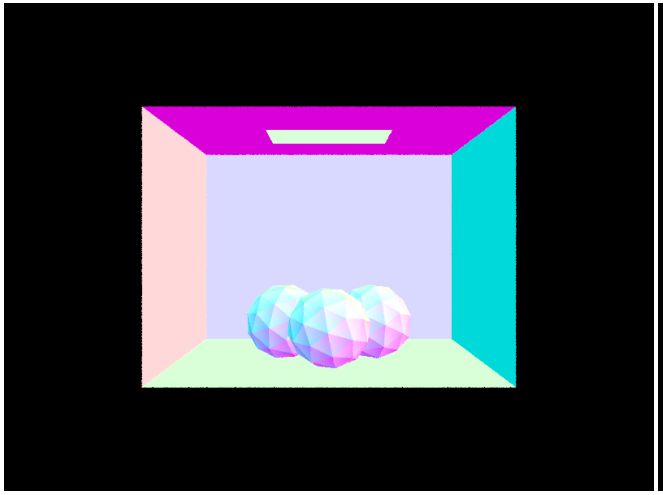
To generate a ray, we require an origin coordinate (i.e. the camera position in world space) and a normalized direction vector. To sample every pixel via ray casting, there are three mappings that must occur to determine the proper ray direction for each pixel. Given a pixel coordinate, the first map is from pixel coordinate frame to a normalized coordinate frame achieved essentially by scaling the axes. The second map takes the normalized coordinate frame to the image frame which defines the field-of-view captured in the scene. This is achieved through a combination of scaling and translation perpendicular to the camera's Z axis. The goal of these first two transformations is to effectively 'superimpose' the pixel plane onto the image image so that we know how to draw a ray from the camera origin through the field-of-view for a given pixel. Multiple sampling for a single pixel can be performed by perturbing the ray direction within the extent of the pixel area. Once the ray direction for the pixel is determined, we apply the camera-to-world transformation on this direction vector so that the ray propagates correctly in the scene. The origin of the ray is also translated to match the camera origin.

Primitive Intersection:

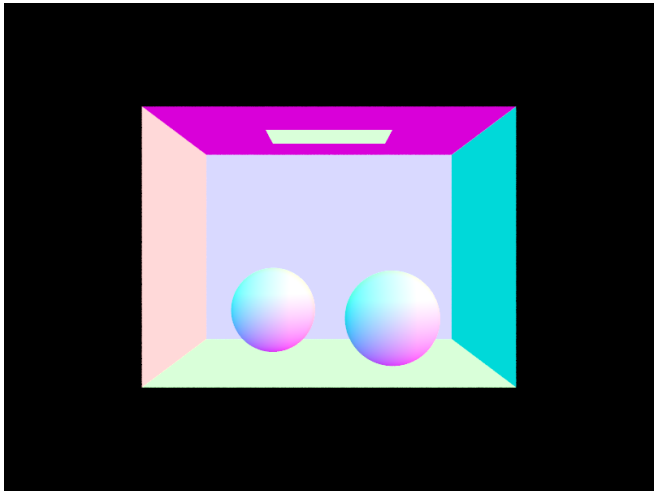
Determining intersections between rays and primitive geometric objects required using implicit vector definitions of the primitive geometries themselves. Intersection with a triangle was implemented using the Moller-Trumbore Algorithm. In short, this algorithm combines Barycentric coordinates and an implicit definition of a plane to determine whether the intersection point resides inside a triangle primitive. By ensuring that the Barycentric Coordinates were all within the range [0,1], and verifying that the intersection t-value was within the maximum and minimum bounds of the scene, I could confirm an intersection with a triangle primitive. The checks for the sphere intersection were more involved. Firstly, the intersection t-values were identified by solving a quadratic equation using the quadratic root formula. Ensuring that the discriminant in was non-negative served as the first check as this would imply that the solution has at least one real root. If so, then checking if either of the roots of the quadratic fell within the max and minimum bounds was necessary. Finally, I would select the minimum root if both roots satisfy the previous conditions.



Intersection with Triangle Primitives



Intersection with Triangle Primitives



Intersection with Sphere Primitives

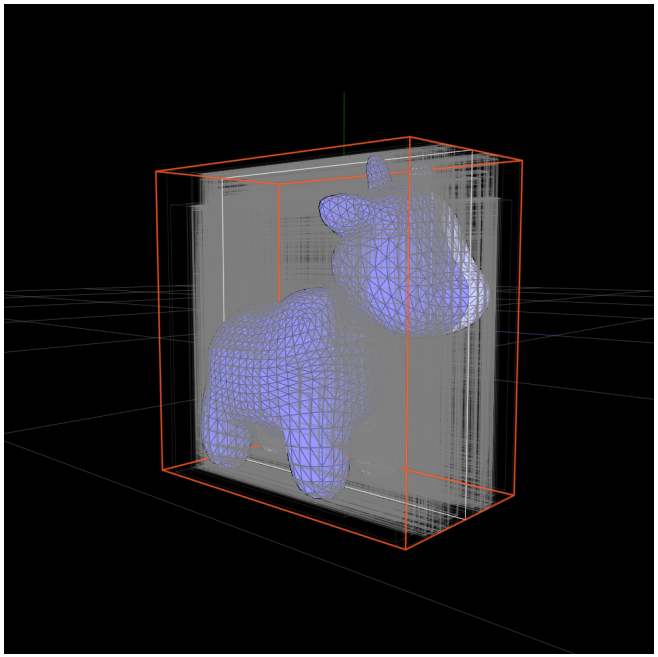
Part 2: Bounding Volume Hierarchy

The heuristic used here to determine the BVH split point drastically improved the ray-tracing efficiency. For a given BVH node, I first computed the mean 3-dimensional position of the centroids of each primitive contained in the node's bounding box. This provided three candidate points along which to split either the x, y, or z axis. Subsequently, I computed the cost of splitting each axis at the mean centroid coordinate corresponding to that axis. The cost function was simply defined as the surface area of the bounding boxes for the child nodes resulting from the selected split point multiplied by the number of primitives in each child node. Then I simply selected the split point that yielded the lowest cost. The images below geometrically show the improvement of incorporating a 3-axis heuristic rather than splitting along the mean x-coordinate alone. In general, the bounding boxes encapsulate the primitives more tightly. This reduces the probability that a ray hits the bounding box of a BVH without hitting a primitive. In other words, the density of primitives per bounding box is drastically improved using the 3-axis heuristic described.

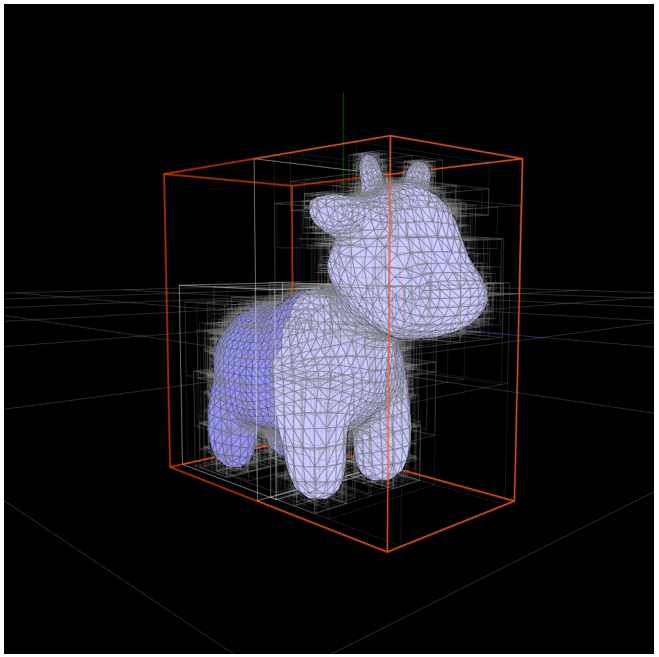
I would also like to justify the choice of using the mean position as the split point rather than the median. There are two factors influencing this decision. First, for a general ray-tracer, we would like it to work well without requiring any a-priori information about the 3D distribution of primitives. Thus, a reasonable assumption is that for an arbitrary scene, the probability of finding a primitive at any point 3D space

follows a uniform distribution. On average, this assumption will be true for the space of all possible 3D scenes and thus provides a BVH subdivision scheme that is robust to a wide variety of scenes. Under the assumption of a uniformly distributed set of primitives, the mean centroid location and the median centroid location along the x, y, and z axes are approximately equal. Thus, over many renders the BVH tree will be balanced. Second, implementing the median position split requires sorting the primitives along each axis first. Given the format of the skeleton code, this would have required running a sort at each step in the recursion thus increasing the time cost for generating the BVH with little improvement in the tree balance given the uniform primitive distribution assumption above.

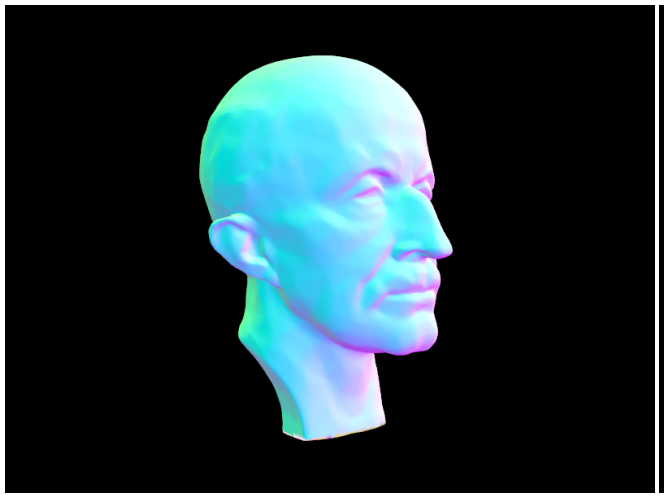
The impact of a BVH on computational efficiency is immense. The cow rendering below completed after 38.1913 seconds without the BVH compared to 0.0183 seconds with the BVH. Moroever, the average number of ray intersection tests was approximately 942 without the BVH compared to only 6 interesction tests with the BVH implemented. The benefits of this organization of the primitives also stems from its appealing computational complexity. Without using a BVH, the average number of interesction tests per ray is O(n) while for a BVH the computational complexity is O(logn) where n represents the number of primitives in the scene. This means that mesh complexity and resolution can be scaled without seriously impacting the rendering time.

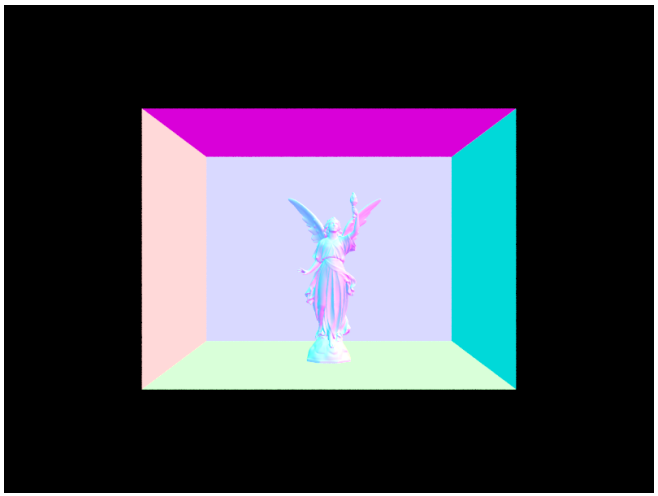


Naive Bounding Box: Single-axis Splitting



Heuristic Bounding Box: Three-axis Splitting





Part 3: Direct Illumination

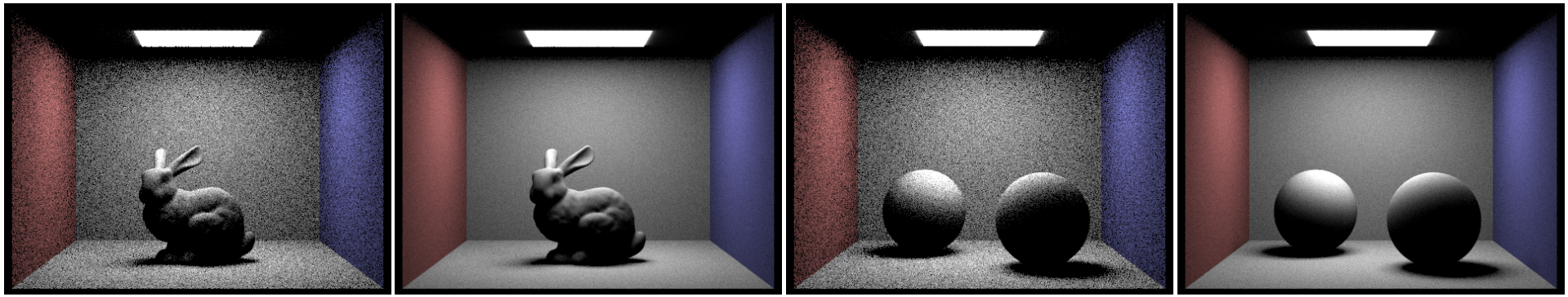
Direct Lighting considers the illumination produced by single-bounce light rays (i.e. rays that originate at the light source and bounce one time or less on the scene objects before arriving at the image plane). Two implementations for direct lighting were explored in this project: Uniform Hemisphere sampling and Importance sampling. Fundamentally, these implementations differ in the way that they sample the incident light ray directions for approximating the reflection equation using Monte-Carlo integration. For both implementations, we consider the inverse light path of a ray starting at the camera and flowing into the scene. Upon intersecting an object in the scene, the radiance that the camera would observe leaving this intersection point depends on the Bidirectional Radiance Distribution Function (BRDF) of the

object's surface and the irradiance arriving at the intersection point. In the Monte-Carlo estimate, we sample several fleeting ray directions in the object coordinate space and propagate new rays in these directions until they reach a light source. If they are obstructed, they are considered shadow rays and contribute nothing to the integral. With Uniform Hemisphere sampling, the direction of the explored rays is not guaranteed to point towards the light source. Hence many of the explored rays have no chance of contributing nothing to the reflection equation integral. In contrast, explored rays generated through importance sampling are guaranteed to point in the direction of the light source. These rays are only discounted if there is an obstruction so the overall irradiance is characterized better with the same number of samples.

In this project, we have only implemented a BRDF for diffuse Lambertian reflection. Hence, the BRDF is for diffuse surfaces is independent of the incident and outgoing ray directions. Therefore, the only terms in the reflection integral that do depend on the incident ray direction are L, the incident radiance, and the Lambertian cosine factor. These quantities were thus computed once the ray direction in object space was sampled.

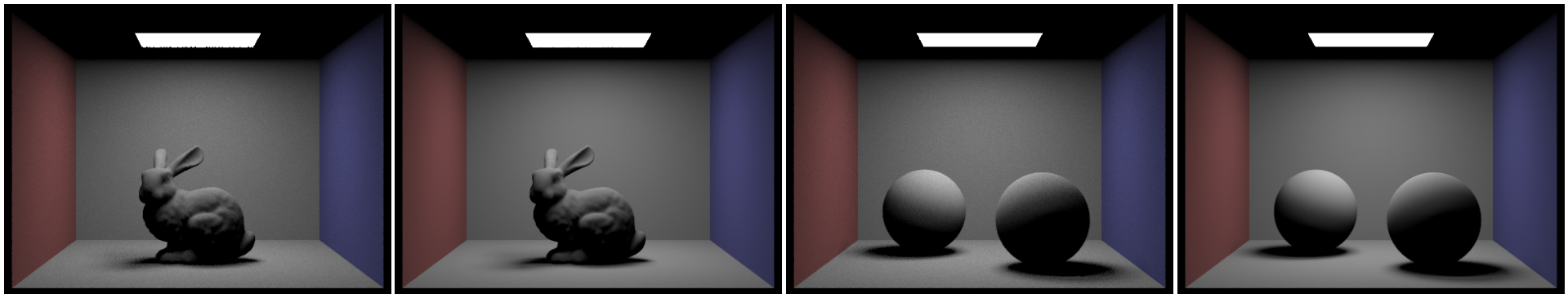
The direct illumination images shown below compare renders involving the Uniform Hemisphere sampling and Importance sampling techniques described previously for different numbers of pixel samples and light samples. Noise in the renders is substantially reduced when using importance sampling. This demonstrates the effectiveness of the importance sampling technique in producing drastically higher render quality for the same number of samples.

Uniform Hemisphere Sampling Renders



16 samples/pixel | 64 samples/pixel | 16 samples/pixel | 64 samples/pixel

Importance Sampling Renders



16 samples/pixel | 64 samples/pixel | 16 samples/pixel | 64 samples/pixel

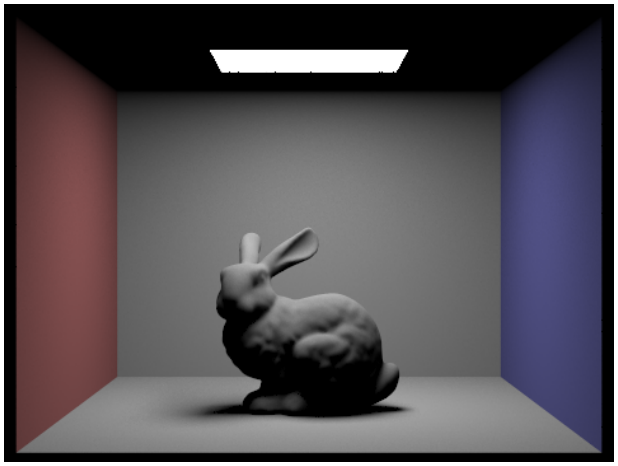
Part 4: Global Illumination

Indirect Illumination considers contributions from light bouncing between non-emitting objects in the scene. At a high level, ray-tracing for indirect illumination explores the space of all possible light paths from the source(s) to the camera by exploiting the recursive nature of the reflection equation. As implemented here, the indirect lighting function performs the following in each recursive step:

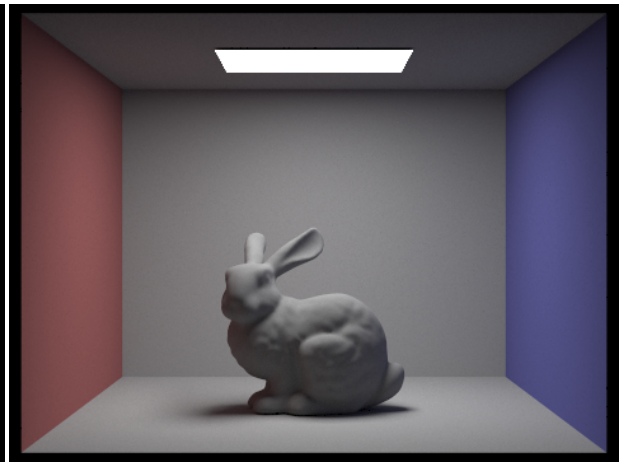
- For a given intersection with a non-emitting object in the scene, collect the direct-lighting contribution at the intersection point.
- Then flip a weighted coin determine whether to continue tracing rays – this unbiased probabilistic termination criterion known as the Russian Roulette approach and addresses the possibility of infinite recursion.
- If ray-tracing is to be continued, randomly sample directions fleeing the intersection point and propagate rays in these directions until they reach another non-emitting object.
- Recurse on this new intersection point.

Indirect illumination enhances the realism of the render by accounting for contributions from multiple light bounces. Comparing the direct illumination with the global illumination renderings of the bunny below, this realism is most apparent in the color bleed from the red and blue walls onto subject and in the softer shadowing. The second row of renderings show the contributions from single-bounce (direct

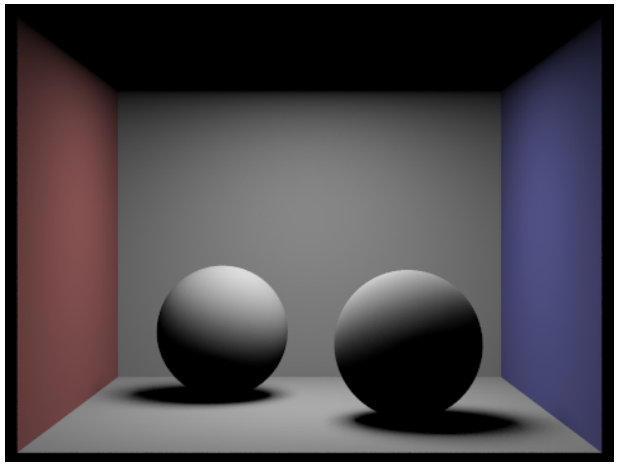
ilumination), two-bounce, and three-bounce light paths in isolation to emphasize their relative importance. The final rendering is the sum of these images and the zero-bounce contribution which represents ray paths going directly between the area light source and the camera.



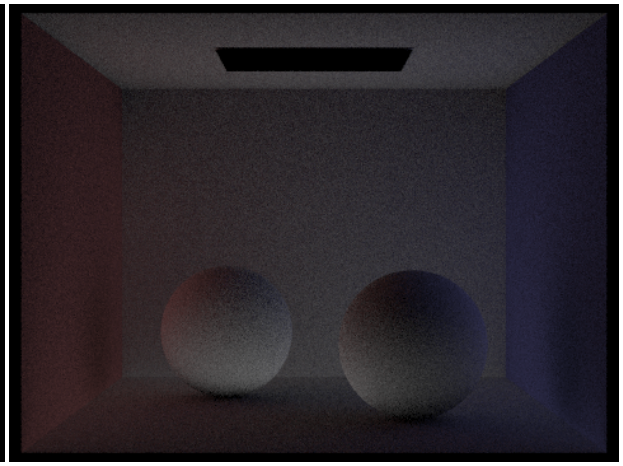
Direct Illumination



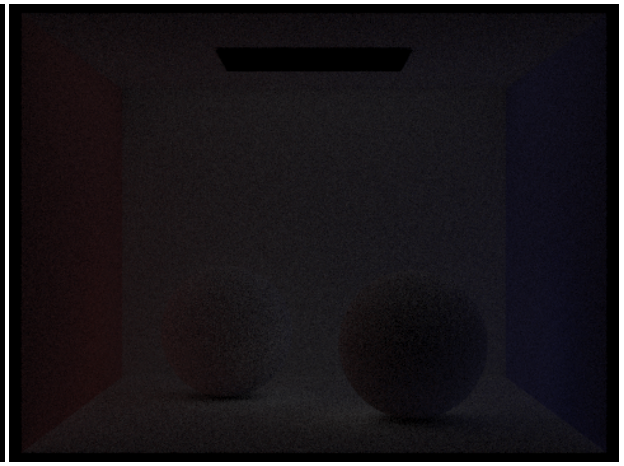
Global Illumination



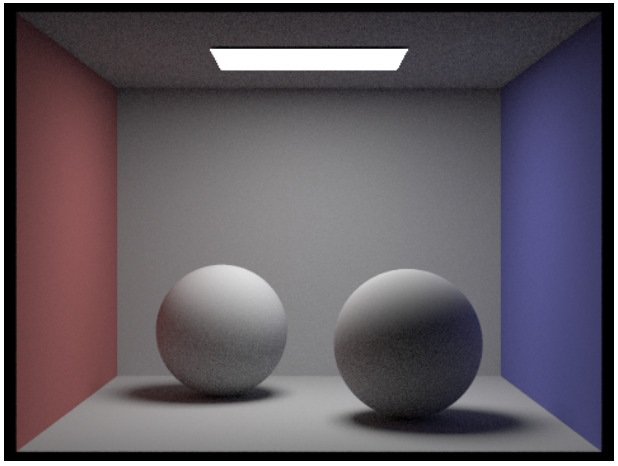
1st Bounce Contribution



2nd Bounce Contribution



3rd Bounce Contribution

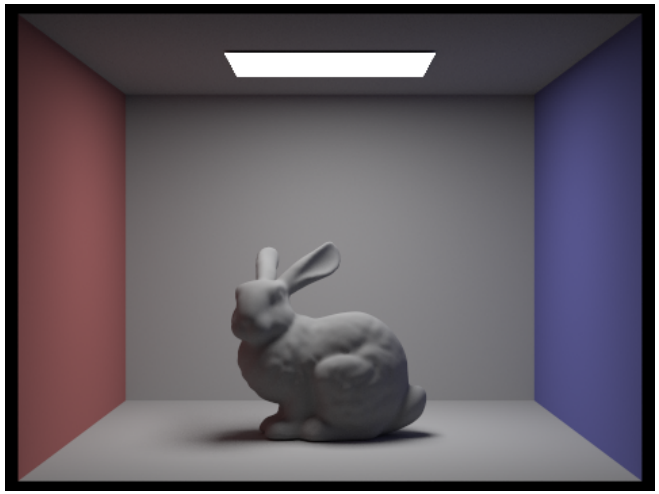


Global Illumination

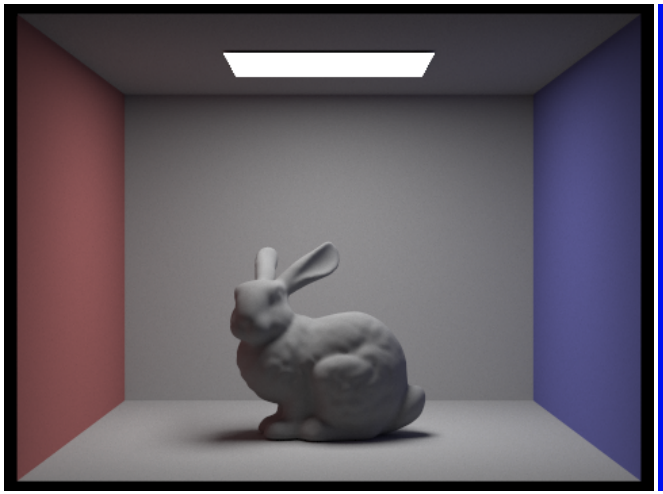
Part 5: Adaptive Sampling

Adaptive sampling intelligently terminates the number of rays processed per pixel based on the convergence behavior of the pixel's illuminance. Pixels that seem to have converged to a stable value require no additional sampling to render the scene with high fidelity. To implement adaptive sampling, I modified the raytrace_pixel() function to terminate its sample-averaging loop before the maximum number of samples if the conditions for illuminance convergence were met. These convergence conditions relate the average illuminance, the variance in the samples, and z-scores by assuming the probability distribution of illuminance in the pixel footprint is Gaussian. Implementing this sampling technique improves computational efficiency as it optimizes the amount of pixel super-sampling. This method is loosely reminiscent of Mipmap architectures in that it marries the variation of information within a single pixel's scene footprint with the degree of super-sampling required for that pixel.

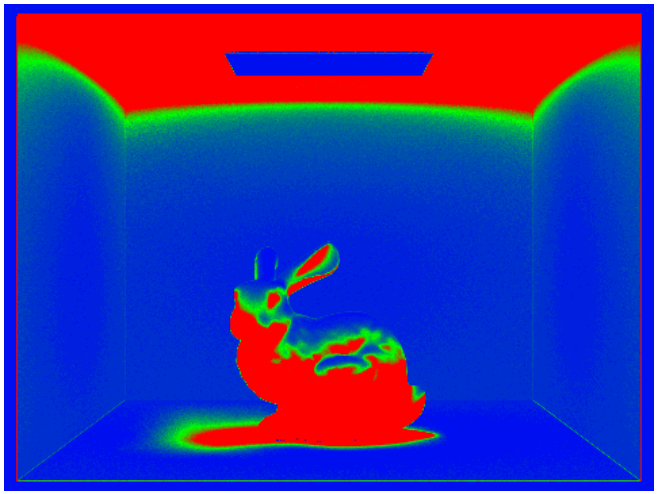
The image rendered with adaptive sampling shown below featured a 2048 sample/pixel upper-bound, a ray depth of 5 for global illumination, and a sampling batch size of 64 rays. Observing the sample rate map, regions in the scene with greater surface complexity and more pronounced contrast gradients, like the bunny, require more samples per pixel. This is because the variance of the samples in these regions is higher and thus takes longer to converge. With adaptive sampling, the render time was 296.2882 seconds, requiring 370,872,646 rays. Without adaptive sampling the render time was 937.0081 seconds, requiring 1,238,167,052 rays. The computational advantages of adaptive sampling are clear. The results described here demonstrate that adaptive sampling reduces the number of rays processed by nearly an order of magnitude while sacrificing virtually nothing in render quality. Comparing the two final images rendered with and without adaptive sampling we can see that both methods yield nearly identical outputs.



Without Adaptive Sampling



With Adaptive Sampling



Adaptive Sampling Rate Map