Assignment 3: PathTracer

Raymond Ly

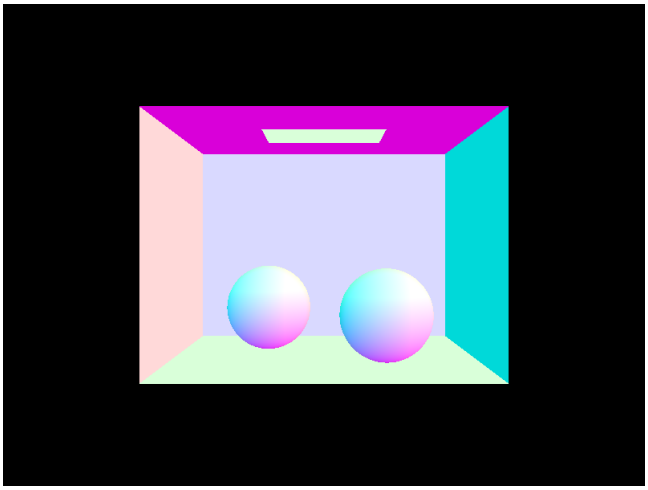
Part 1: Ray Generation and Scene Intersection

To generate rays in our world space, we have to convert the provided camera/sensor space from our Field of View coordinates to our sensor coordinates with a function defined as

```
f(t) = c + (d-c)/(b-a) * (t - a)
```

which takes a value, t in range [a, b] and converts it to a corresponding value f(t) in [c, d]. With this conversion, we transform it again to get a World space vector, with which we normalize to get a ray direction. Then using the Ray constructor, we have a Ray object with the origin, direction, and max distance of the ray from the pinhole.

With this generated ray, we use Moller-Trumbore Algorithm to determine the time, t, that the ray finds an intersection with object in the world space. We then check if the intersection time is valid, updating an Intersection object if, and only if, it is.



Spheres

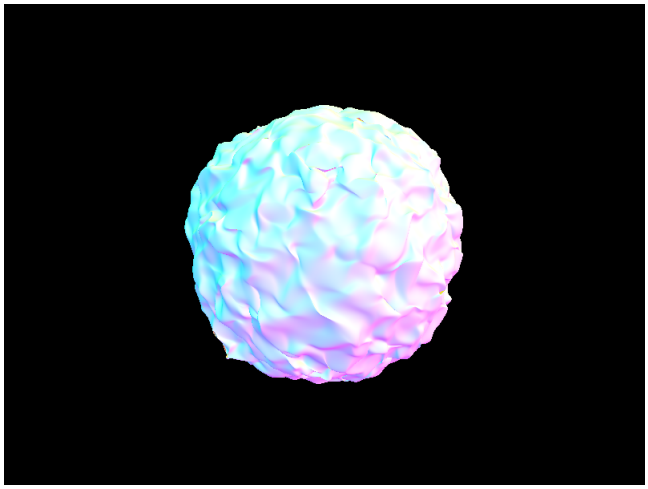
Cow

Part 2: Bounding Volume Hierarchy

We calculate a each of our bounding boxes recursively by generating a tree of BHVNodes. We linearly compute the bounding box with each of the primitives and initialize a new BVHNode. If it is necessary to split (i.e. there are more primitives than the max number of primitives per node), we split the set on an axis, based on the largest dimension of the bounding box we are in and recursively generate more BVHNodes until we reach a leaf node, giving us the smallest possible increment BVHs.

We can calculate whether or not our BVH is intersected by projecting our ray outward to intersect with each individual bound/edge given by the BVH. With that, we have a set of bounds that t intersects the bounding box at. From this set of components, we determine the furthest possible nearest intersection and the closest possible second intersection. After computing these, we update our t1 and t2.

Max Plank



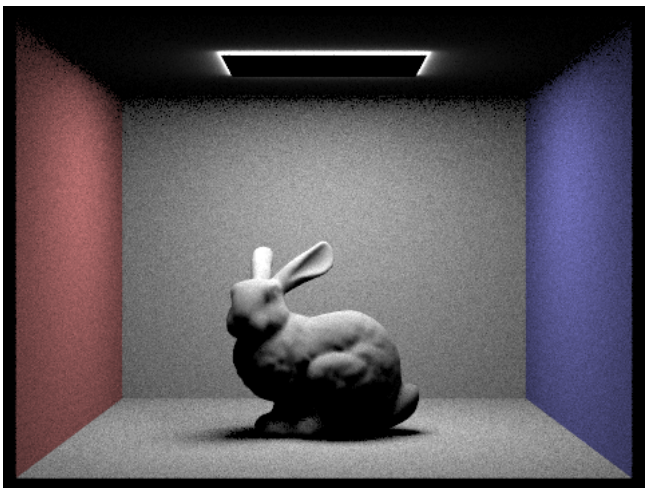
Blob

The above three normal shading renders are generally more complex than the Spheres and slightly more difficult to render than the Cow in Part 1. Without implementing each bounding box, the necessary calculations to compute this volume of normals is far less efficient and can become incredibly cumbersome over time. Focusing on the Cow render, on a standard machine, may take upwards of about 2 minutes to fully render. With our implementation of BVHs, we reduce the amount of time necessary for

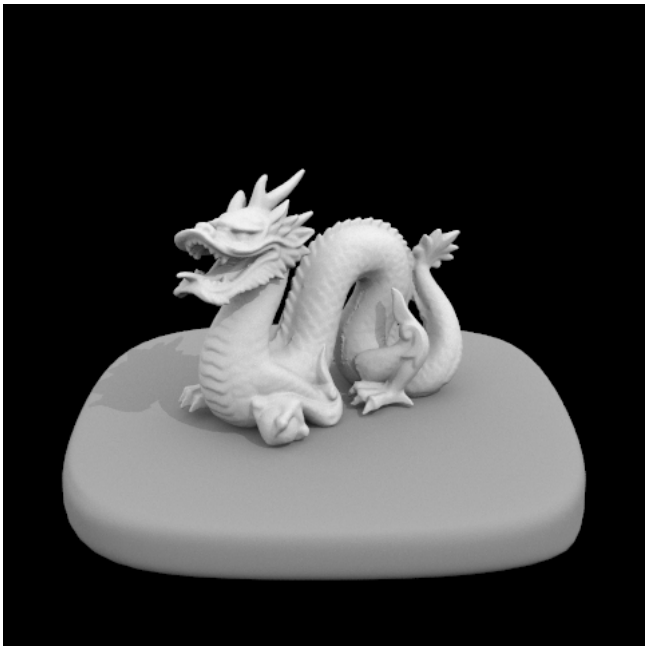
rendering by about 3/4, allowing us to output an image in just 30 seconds. More complex surfaces, like the Blob as shown, took about 10 minutes on my personal machine to output a visual, but by using the bounding boxes, the render time was cut down to just 2-3 minutes.

Part 3: Direct Illumination

In estimate_direct_lighting_hemisphere, we iterate for num_samples iterations, each time calling bvh->hemisphere_sampler() to get a vector in Object space. As with before, we convert this vector into World space and generate a corresponding ray. We check against the bvh whether or not our casted ray has intersected an object. If the ray does indeed intersect an object, we calculate the irradiance of the object by taking its incoming radiance and scaling it by the intersection bsdf and a cosine factor. We then accumulate that value into an overhead outgoing total irradiance varaible. Since we want an average, at the end we divide the calculated total irradiance by the number of samples we considered.

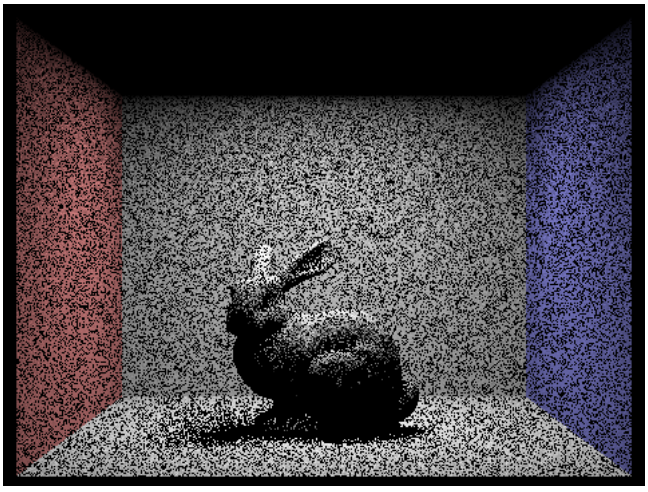


Bunny rendered using Lighting Hemisphere (64 samples, 32 light rays)

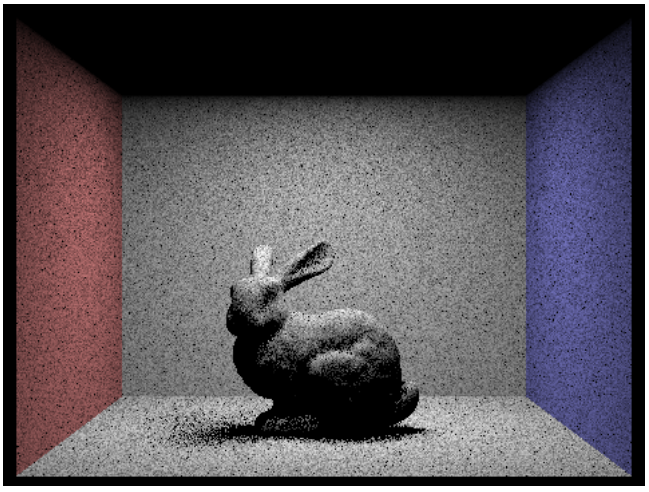


Dragon rendered using Lightng Importance (64 samples, 32 light rays)

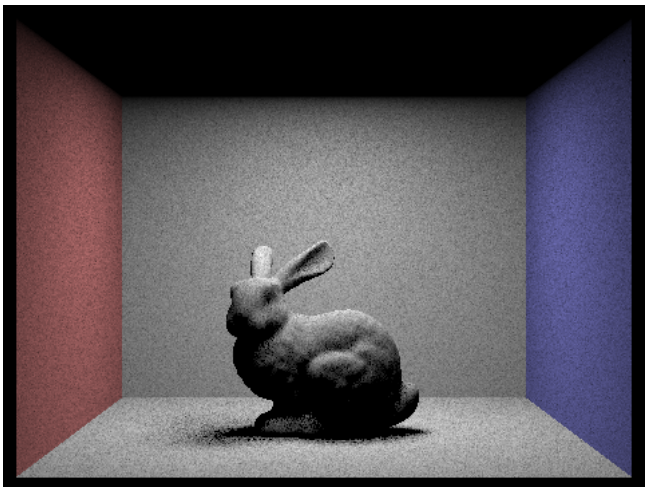
Similarly, we constructed a function for directly sampling over the lights in our scene. Instead of iterating over the number of samples we want, we instead consider each individual light source in our scene. At each light, we consider num_samples sampling rate, effectively subsampling some amount of rays for each light. From each sampled vector, we have to convert it from World space into Object space and generate a ray from our hit point and check if the ray intersects any object. If it does, similarly we accumulate that intersection's irradiance as above into a temporary variable. When directly sampling light sources, we have to take the average irradiance PER light source, so we average this temporary irradiance by the number of samples we took from it and accumulate those into an overarching total irradiance.

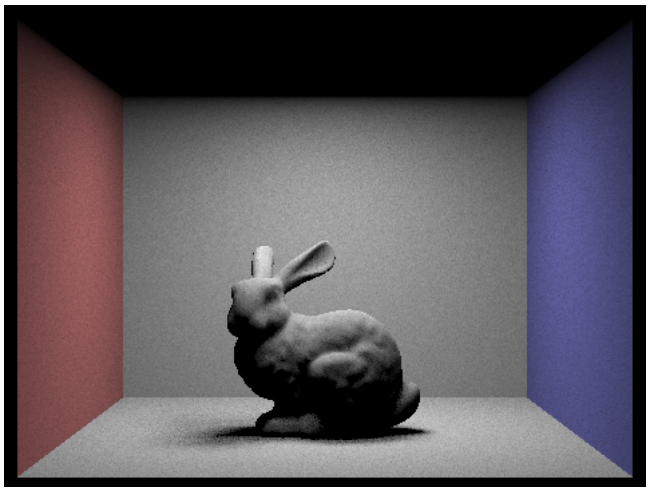


Bunny rendered using Lighting Importance at 1 Light Ray



4 Light Rays

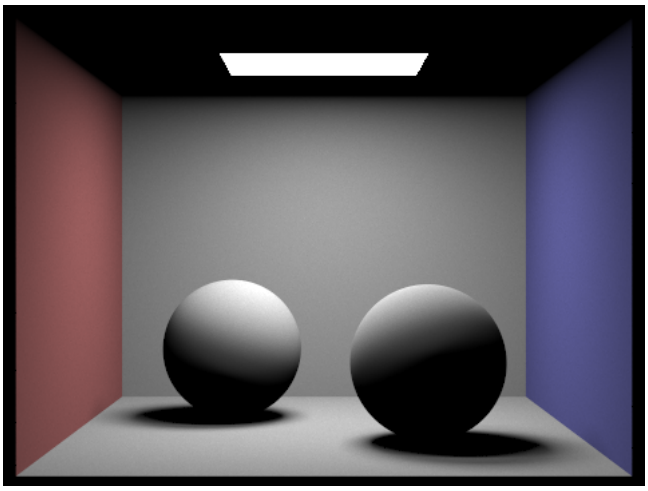

16 Light Rays

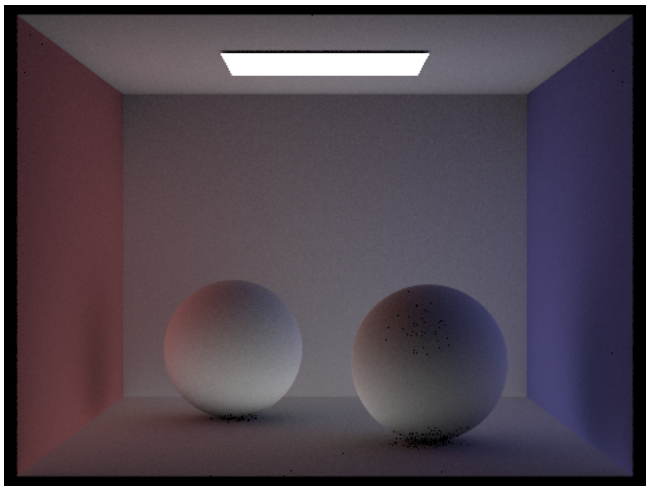

64 Light Rays

Comparing these two methods of sampling, we immediately notice there is a major difference in the amount of noise we get at the same sample rate/settings, as depicted above between the Bunny and Dragon. However, it is also possible for each of them to converge to the same image. It seems as though through hemisphere sampling, we have a more gradual convergence, causing noise to be far more prominent at the edges of our bounds to persist for far longer. Meanwhile, lighting importance sampling cleverly checks whether or not an intersection is being blocked by another object, making rendering slightly more efficient at smaller sample loads. We also notice from each step of the Bunny render in the second figure that it seems importance sampling converges in far fewer iterations, showing most lingering noise at only the darkest of areas on our scene.

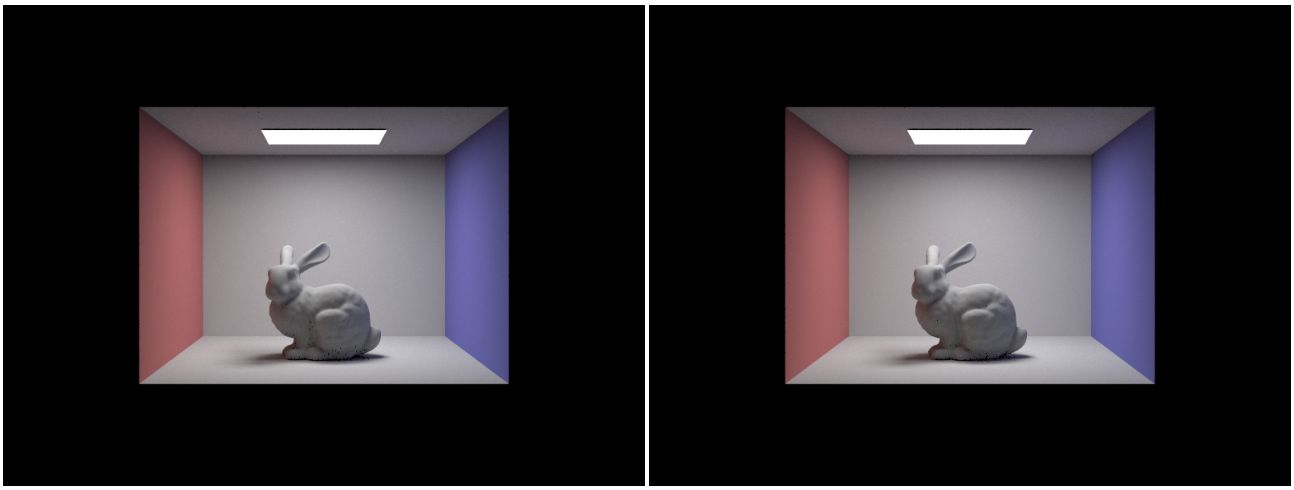
Part 4: Global Illumination

For our implementation, to calculate indirect lighting, the idea is essentially to continuously cast rays for ray_depth number of iterations until we may return some irradiance measurement. We do this in a recursive manner that covers (3) cases; ray_depth is 0, ray_depth is 1, and ray_depth is greater than 1. the cases for 0 and 1 depth are self explanatory in that we either cast no rays or just one ray to sample irradiance. Otherwise, we continue in the same fashion as with Direct Illumination by casting a ray for each intersection and calculating irradiance at that point. However, there is an extra step in our Global Irradiance function, and that is to leave whether or not we continue to expand our function up to randomness. Then, after accounting for this randomness in our calculation of irradiance, as well as some conversion factors, we accumulate this into an overhead total illumination variable.

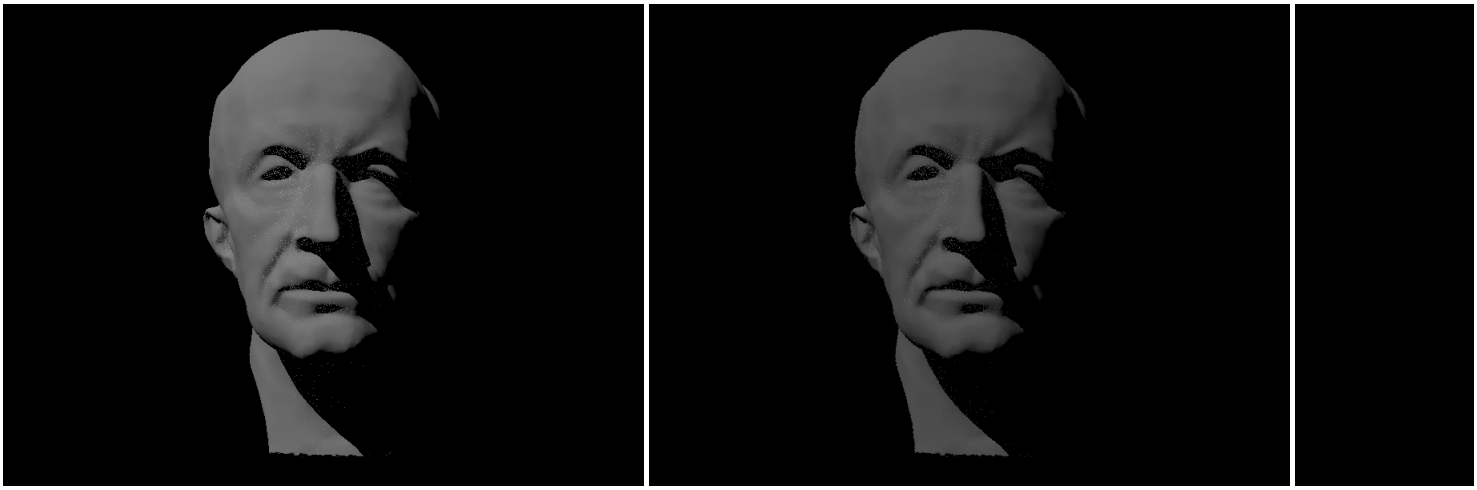

Spheres rendered with Direct Lighting


Spheres rendered with Indirect Lighting

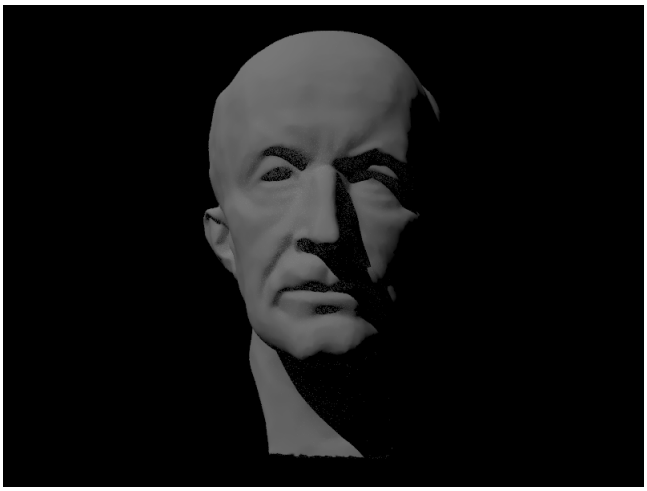
Bunny rendered with max_ray_depth = 0 max_ray_depth = 1



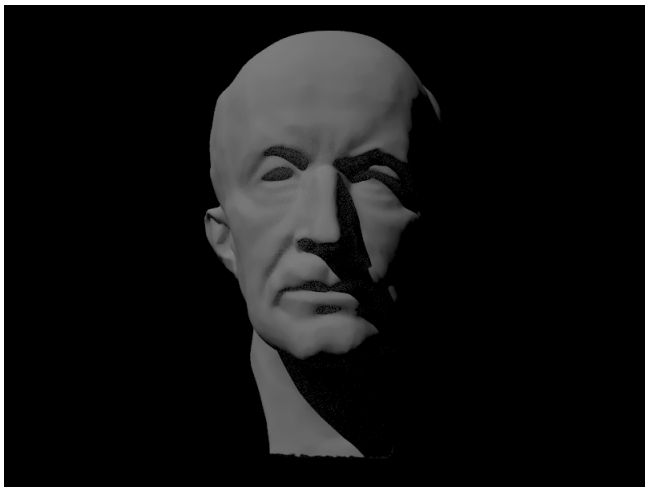
max_ray_depth = 3 max_ray_depth = 100



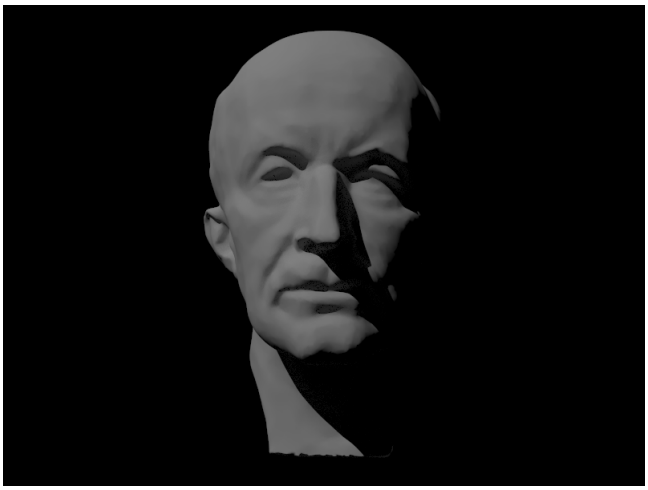
Plank rendered at 1 sample per pixel, 4 light rays 2 samples per pixel



8 samples per pixel



16 samples per pixel



1024 samples per pixel

Part 5: Adaptive Sampling

As an optimization to our code so far, we implement adaptive sampling, allowing our sampling function to break out of the current iteration of a sample in favor of moving on to the next sample. We do this by keeping track of some value, samples_per_batch. With samples_per_batch, we are able to check whether or not our irradiance measure has sufficiently converged. To do so, we construct a 95% confidence interval and check it against a factor of tolerance. We then get the series of equations:

$$I = 1.96 * o/\sqrt{n}$$

$$s1 = \sum x$$

$$s2 = \sum x^2$$

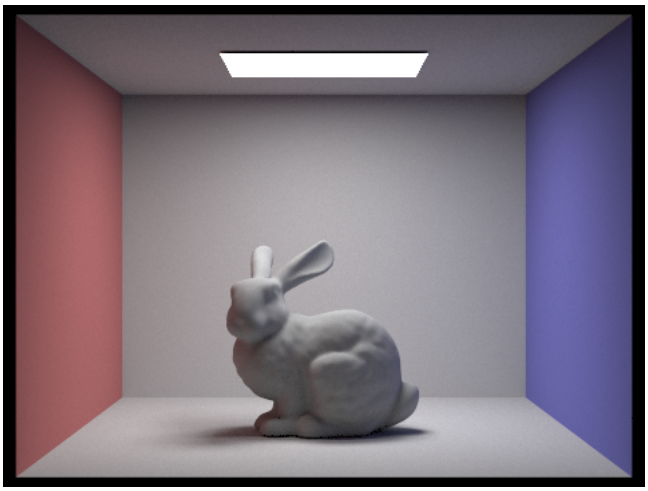
$$o^2 = 1/(n-1) * (s2 - s1^2/n)$$

$$u = s1/n$$

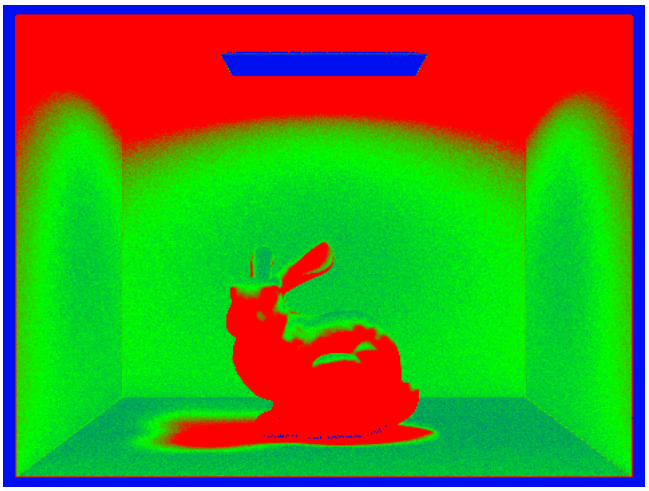
where n is the current number of samples we have taken so far. With this, we check

$$I \leq Tolerance * u$$

If this is the case, we immediately return the currently measured irradiance level and move on to the next iteration of our samples.



Bunny rendered at 2048 samples per pixel, 1 light ray, 5 ray depth



Convergence rate map for Bunny