Assignment 3: PathTracer

Vincent Tantra

For this project I built a path tracer that used ray and lighting concepts in order to render more realistic visuals. The concepts used in this project include ray generation, intersections, different methods of lighting, and adaptive sampling!

Part 1: Ray Generation and Intersection

Part 1 consisted of casting rays from a source onto an image plane. We generate rays from a single point in space (similar to the concept of pinhole camera) and create an output that consists of the image found by those rays.

The first step is to generate said rays. We first implemented the generate_ray function, which, as named, generates rays from a camera, or a single point in space. We use given parameters x and y to determine the rays direction from this origin; however, x and y are given within the range 0 to 1, so we also need to map them to the respective coordinates in the image plane using interpolation, which redefines the origin point at the bottom left of the image, and the (1, 1) point at the top right.

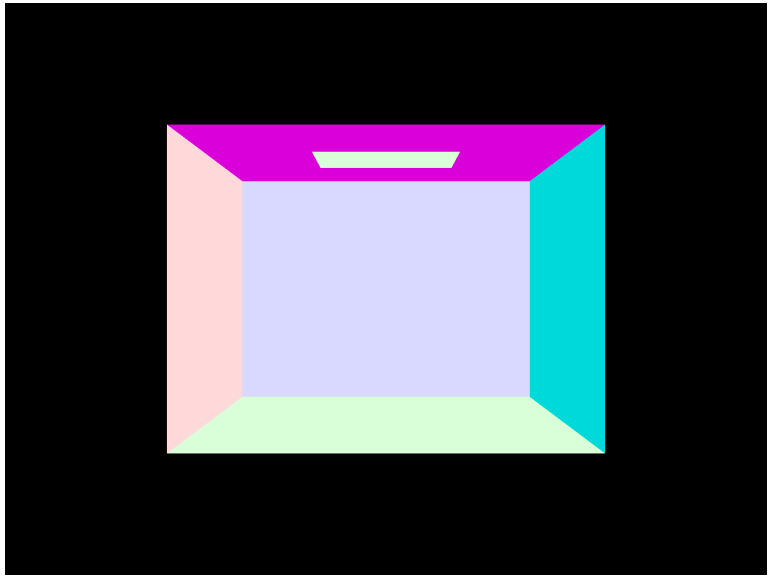
Then, we use these rays in the function raytrace_pixel. We implement 2 different scenarios here. We can cast a singe ray through the center of the pixel, or we can sample ns_aa rays through the pixel at random locations, ns_aa being a provided parameter.

Once we defined how we generate and cast rays, we need to define the objects in the scene that they interact with.

We use two primitive objects in this project: triangles and spheres. First, we implement triangles!

Triangle primitives are implemented using the Möller-Trumbore algorithm, which uses concepts similar to those used in our first project. We combine the barycentric formula from Project 1 with a formula for an intersection between rays and triangles: $P = O + tD$, a simple vector parametric equation consisting of time, Directio, and Origin. By combining these equations, we essentially get a final formula that allows us to derive when and where a ray intersects with a triangle, if at all. We use Cramer's Rule to help with the linear algebra; this rule allows us to find the solution to the linear equation by multiplying the determinant of the matrix by the diagonal of the 1x3 matrix involved. Thus, after many dot and cross products, we can use some conditions to check if we are left with a ray that intersects the triangle, those conditions being:
- t is between min_t and max_t
- u is between 0 and 1
- v is more than 0 and u + v is less than 1

These conditions basically check if the intersection holds true. And combined with the above algorithm, we are able to have rays interact with triangle primitives.

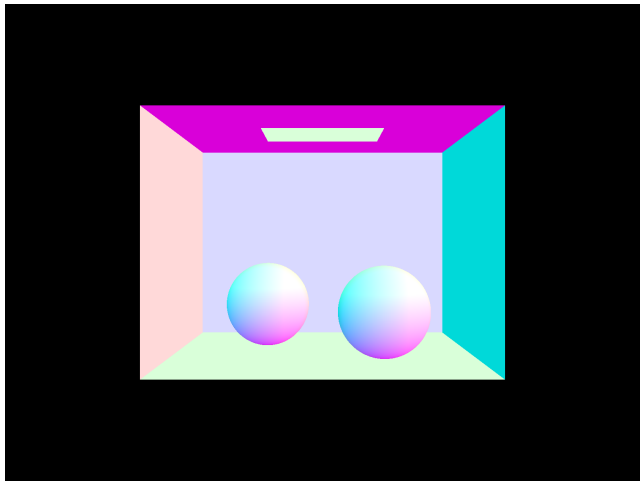


A simple box can be made!

For our second primitive (spheres) we take a different approach. Spheres can be intersected twice by ray, and their intersection is similar to that with how a line crosses a parabola at most 2 points in the 2D space. Thus, we use the quadratic formula. The following variables are the values for our quadratic:
- a = dot product(ray direction, ray direction)
- b = dot product(2(ray origin - sphere center), ray direction)
- c = (ray origin - sphere center)^2 - Radius of sphere^2

3 scenarios exist. We can have either no intersection, 1 intersection (tangent to the sphere), or 2. This is determined by the discriminant of the quadratic formula, so we use that to check our 3 cases.

With that implemented, we can now render spheres inside of our box, as well as simple figures!



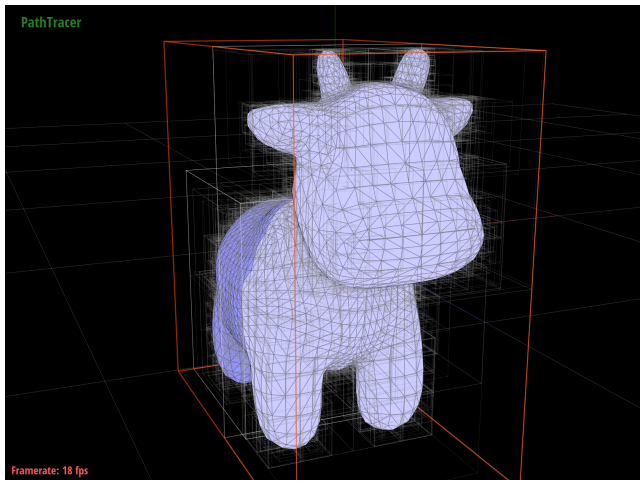
Two pretty spheres!

A cow figurine.

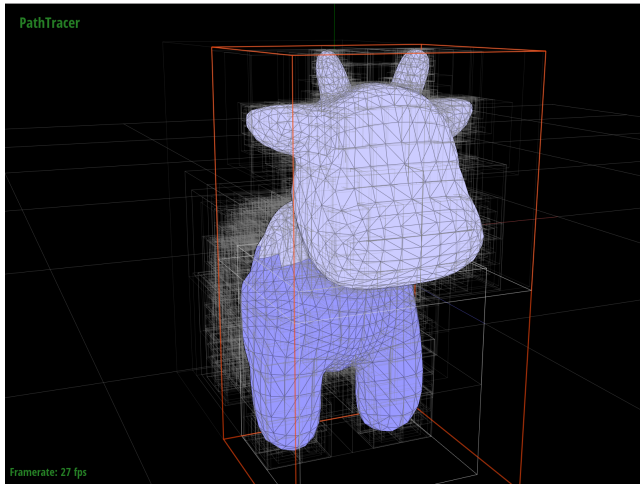
Part 2: Bounding Volume Hierarchy

A Bounding Volume Hierarchy (BVH) puts the objects in a scene into a data structure (a binary tree) of rectangular volumes, using our 3 axis in 3D space as the separation factor. This allows us to test ray intersection with axis-aligned planes instead of brute-forcing every single primitive in the scene.

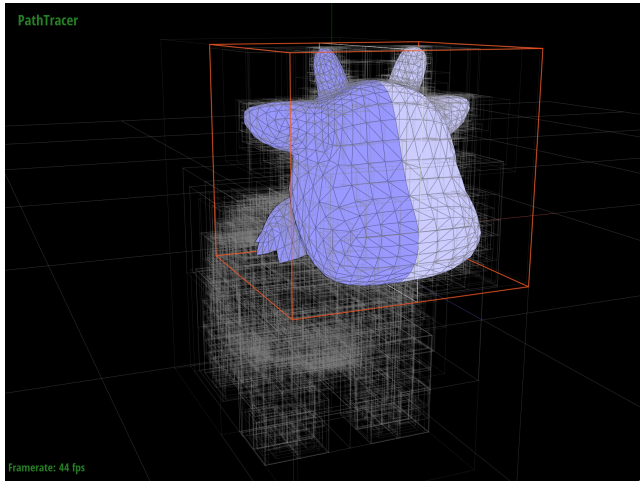
My heuristic for my BVH is relatively simple. I first choose the largest axis (found by using the "extents" of the bounding box), and decide to split the tree using that midpoint of that axis as the separation point. I then separate nodes to the left or right depending on their relation to the split point. In the edge case that all nodes lie on one side, I return the whole thing, since I believe relatively few nodes will ever just be on one side together, and the case of this happening is rare.



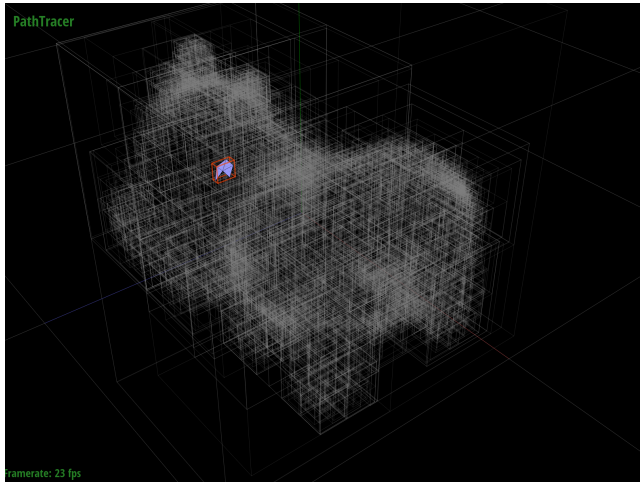
The full mesh of a cow.



We then cut it in half!



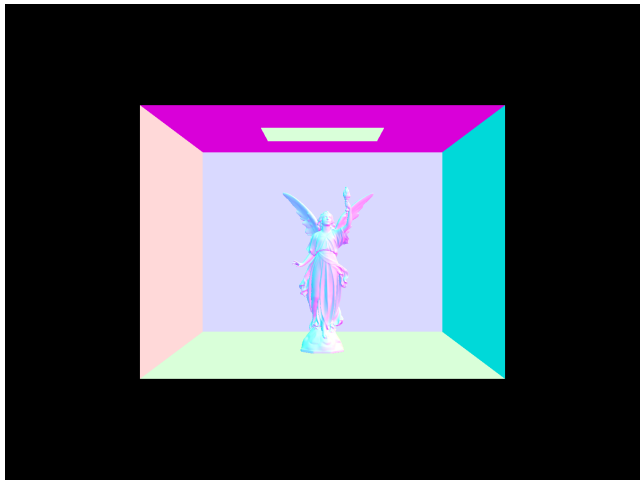
Half again...



Until we end with a single leaf node!

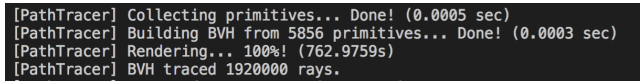
We can now use this to find our intersections in a more efficient way. We first test if a ray hits the bounding box of our BVH. If it doesn't, we stop. If it does, we then need to consider whether we are dealing with a node that has two branches, or a leaf that has a list of primitives to check. If we are dealing with the former, we recursively check for interesections on both the nodes the current node we're at, and return the closest hit value from those calls. If we have a leaf, we check for intersections on each primitive the leaf holds, and return the closest hit. After this was implemented, I was able to render some more complex files!

Such a detailed face!

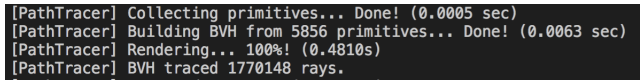


Such a detailed figure!

It's important to note just how much of a speedup we get from impelementing this. The following are 2 screenshots of the statistics from trying to render a simple cow figurine. Notice that we trace far fewer rays, but the speedup is significant. Considering this was a simple figurine that was rendered, without any complex shapes or grooves, such a speedup is essential for using our rays on other objects.



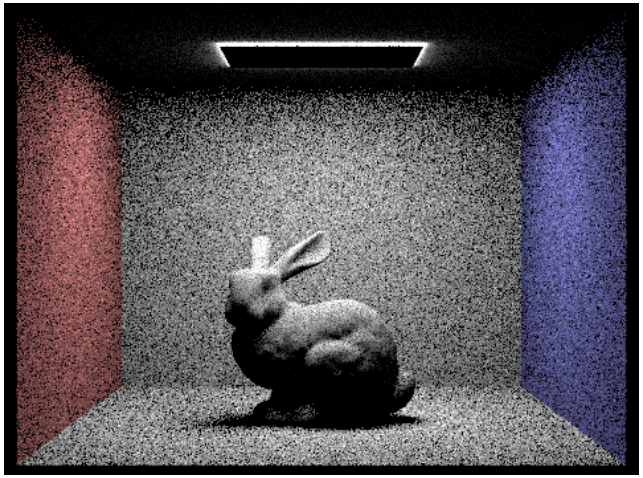
700 seconds!



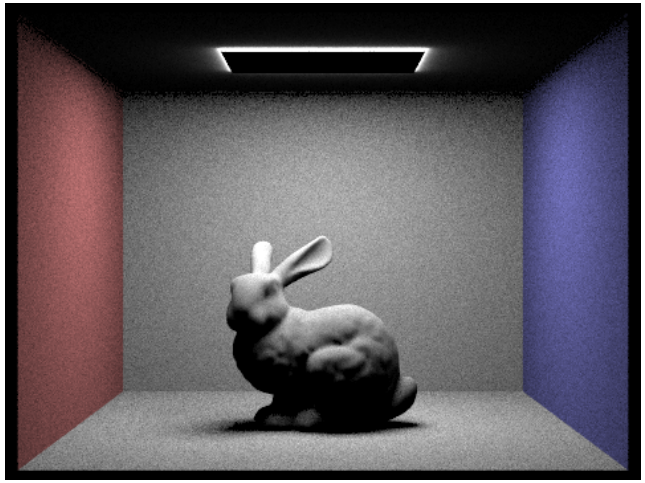
That speed is much more desirable.

Part 3: Direct Illumination

We implement direct illumination in two ways: using hemisphere sampling and importance sampling. Uniform hemisphere sampling samples in a uniform hemisphere around hit_p, our point, and computes the incoming radiance from a light source for each ray that intersects that source. It then converts this radiance into an outgoing radiance using the BSDF of the surface, and finally averages these across all samples. While the implementation is correct (converges to a good render) and simple, renders with less rays tend to be very noisy.

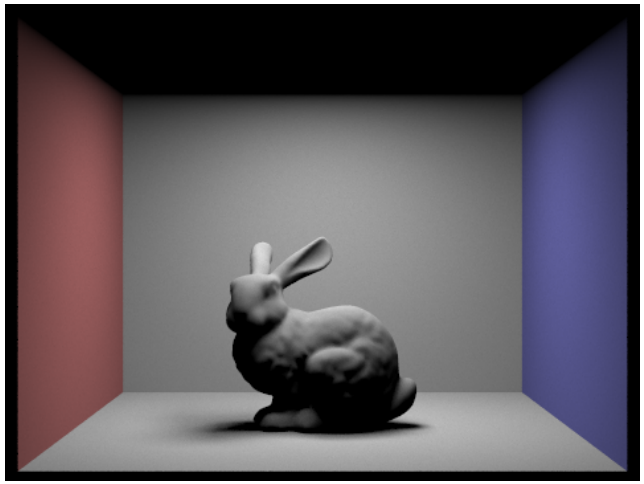


Bunnies are quiet, but this image is noisy.

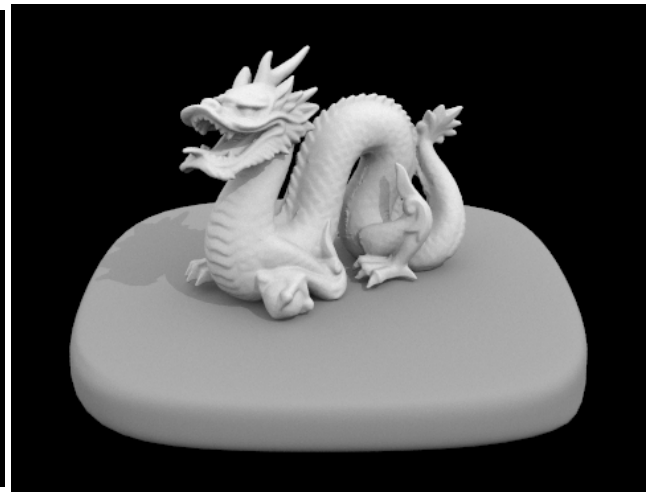


Increasing the number of samples and light rays makes it better.

This is where importance sampling comes in. The difference between importance sampling and hemisphere sampling lies in where samples are taken; importance sampling samples from each light directly, summing over every source in the scene before doing the same radiance calculations as the previous sampling method. We first check if the light is a delta light, meaning we only sample once since all samples would return the same radiance. Otherwise, we take several samples. The resulting image is much smoother.

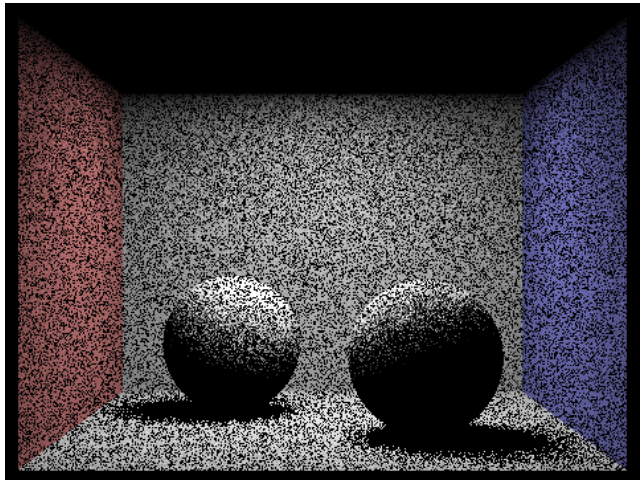


A cleaner rendered bunny.

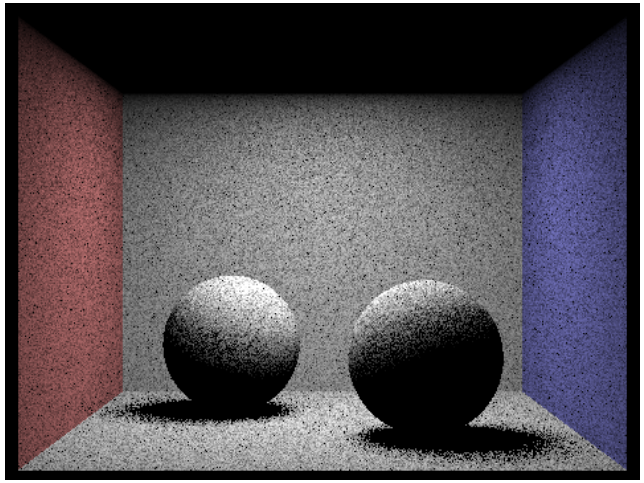


We can also render dragons with the correct shadows.

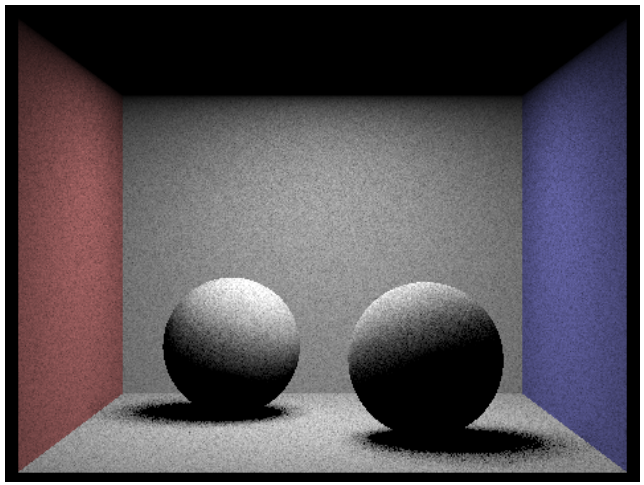
As we can see, importance sampling produces a cleaner image, even with the same amount of rays. We can also alter the number of light rays to produce a more detailed render, as shown below (I take 1 sample per pixel for each of these renders).



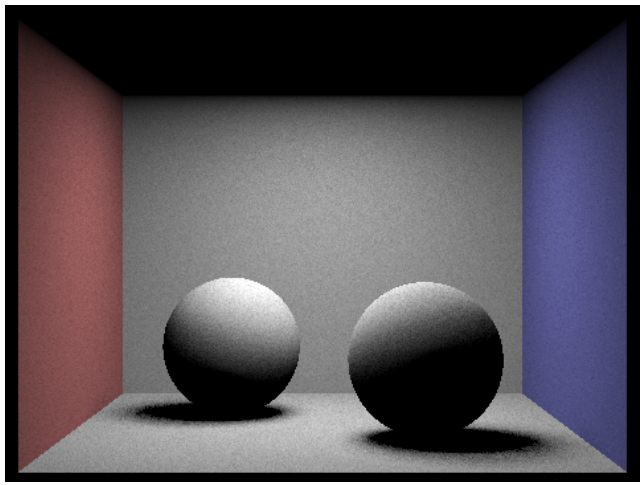
1 light ray sampled.



4 sampled.



16 sampled.



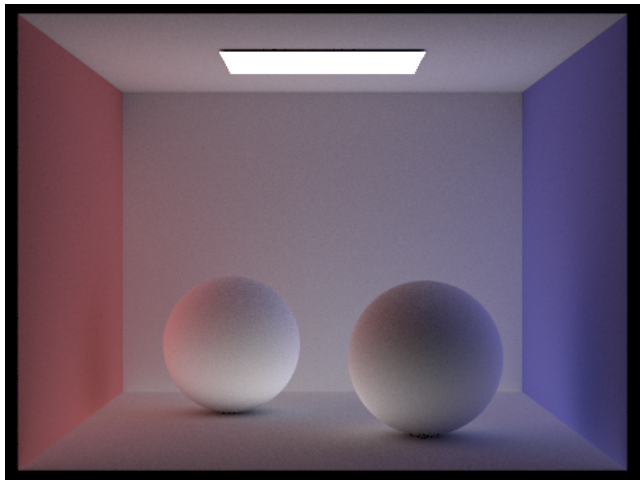
64 sampled. The cleanest image!

Part 4: Global Illumination

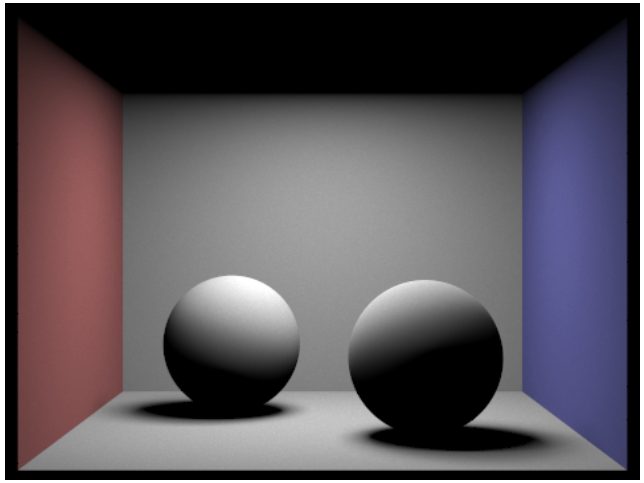
We then implement indirect lighting and full global illumination. This feature implies that some light rays can bounce infinitely, so we need to have a method or estimating their radiance without just falling into an infinite loop. We do that by using several methods, each of which account for a certain case of ray bouncing.

For indirect lighting, we implement a function that computes at least one bounce. This function takes in a starting spectrum and adds to it using the Russian roulette algorithm. At a high level, we have a set probability that curtails how many times we bounce our light. We make sure we bounce the ray at least once if the maximum ray depth is greater than 1, but otherwise, we do several checks that cut off the ray faster if they fail: if the Russian roulette probability tells us to stop, if we've reached the end of the ray depth, or if the maximum ray depth is not greater than 1 (in which case we shouldn't be recursing in the first place).

We also code a function that computes if zero bounces occur, which is just the emission of light from that surface (black if that point is not a light source). Combined, this gives us the result of solely indirect lighting:

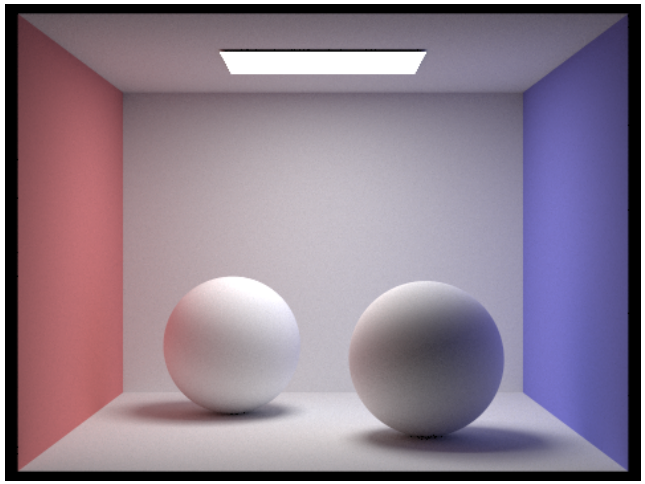


Spheres only rendered with indirect lighting.



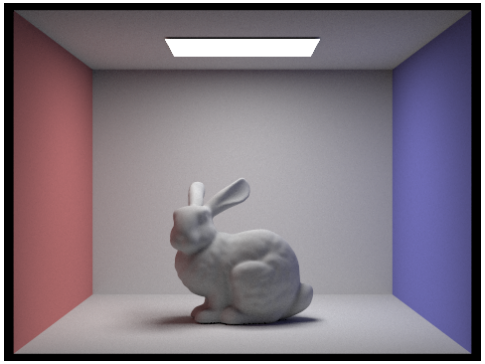
Spheres only rendered with direct lighting.
Notice the contrast between the two!

Finally we combine both indirect and direct illumination to give us global illumination. We do this just by adding all the different bounce functions together, initializing with zero_bounce, starting with one_bounce and recursively adding to that using at_least_one_bounce. Featured are the our balls image and bunny image rendered with global illumination at 1024 samples per pixel.

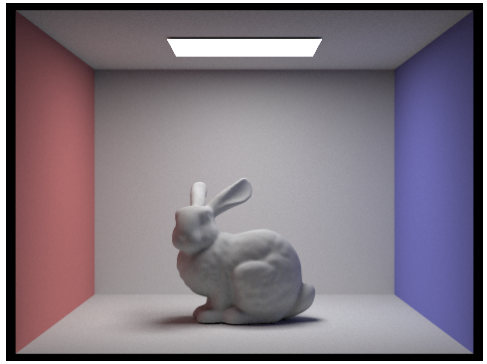


Spheres rendered with global illumination.

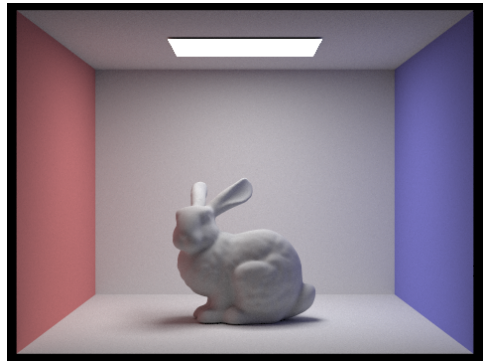
Let's compare the quality of image of the bunny when changing the max ray depth. The larger the max ray depth, the more we allow the light to probably bounce.



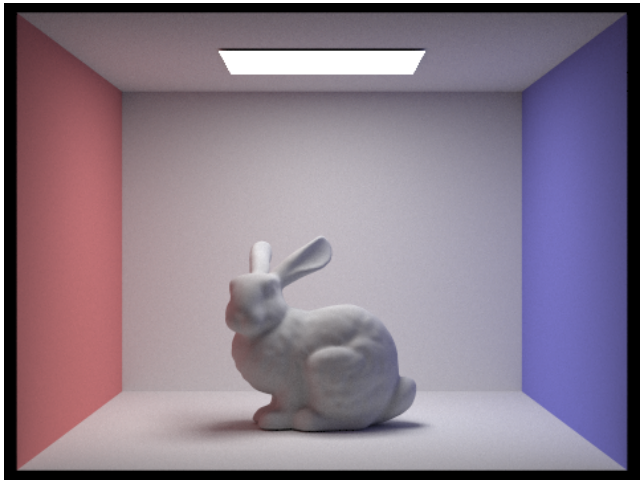
Max ray depth = 0.



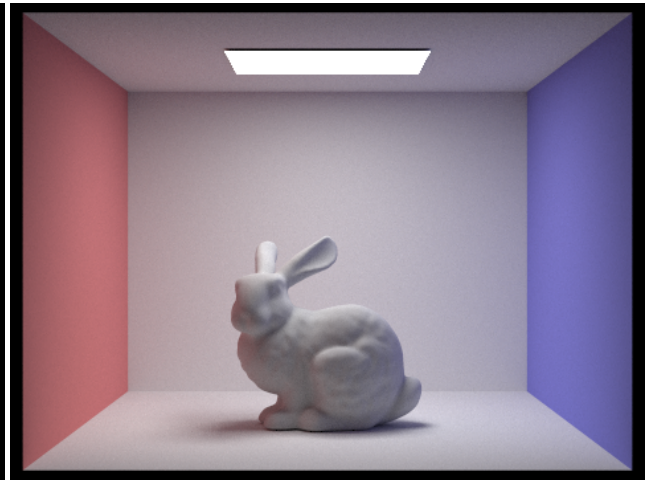
Max ray depth = 1.



Max ray depth = 2.

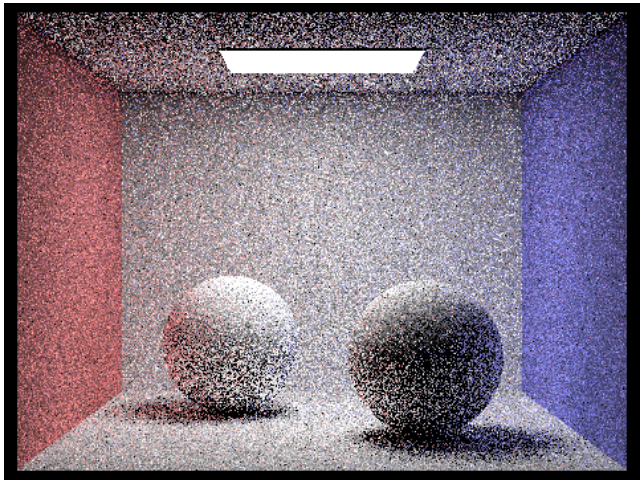


Max ray depth = 3.

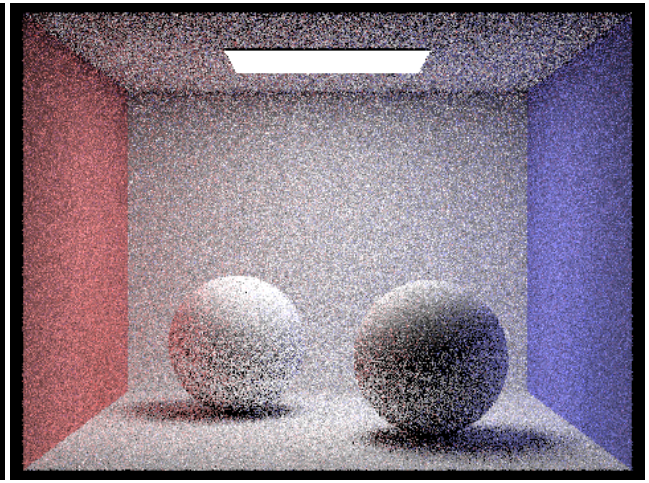


Max ray depth = 100.

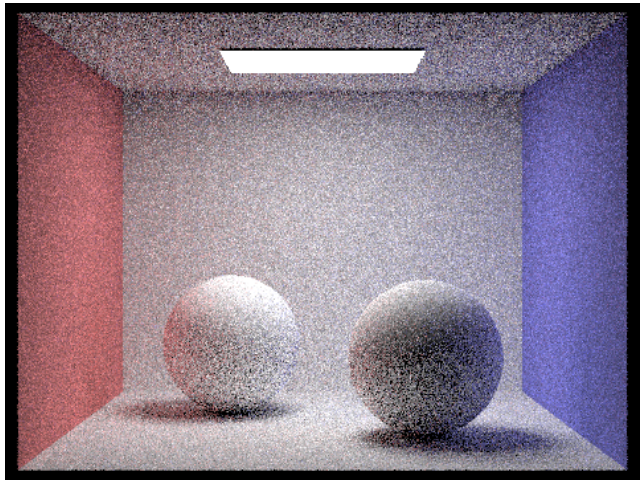
Let's also look at how various sample-per-pixel rates affect the scene. We use the ball scene for this case, and 4 light rays.



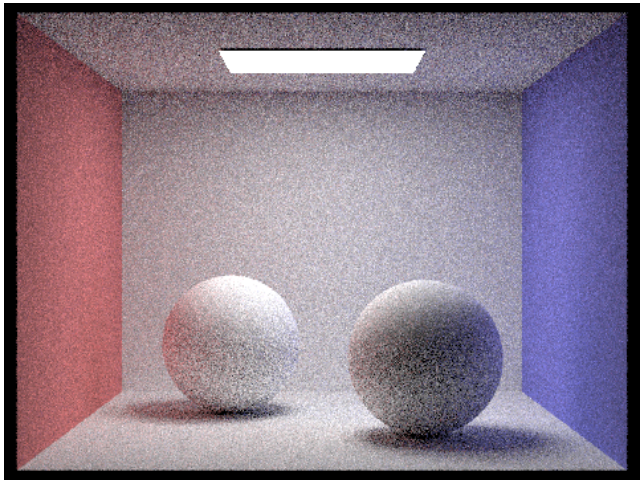
Samples per pixel: 1.



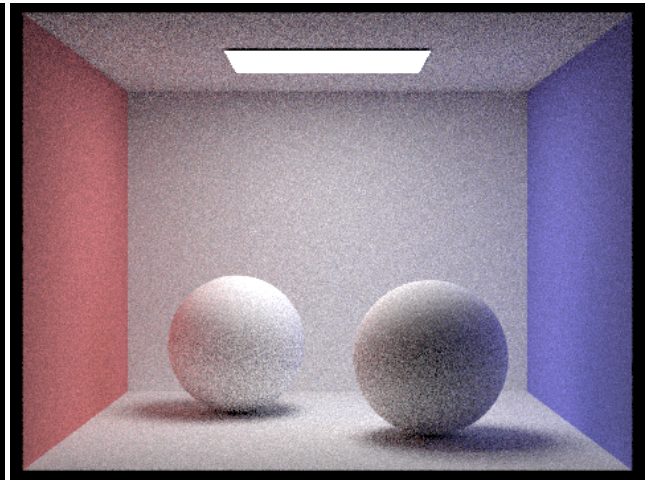
Samples per pixel: 2.



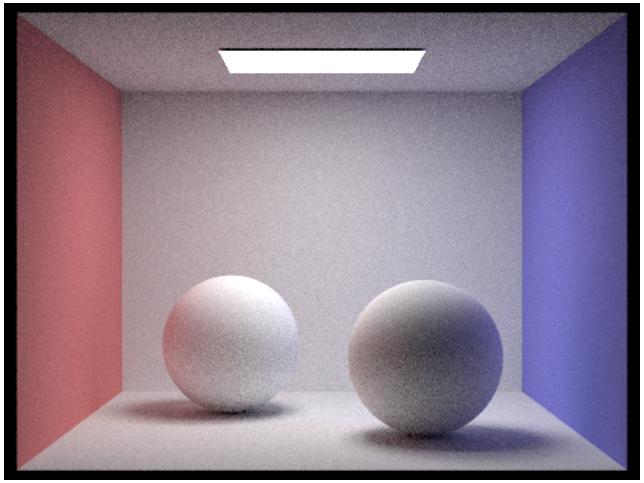
Samples per pixel: 4.



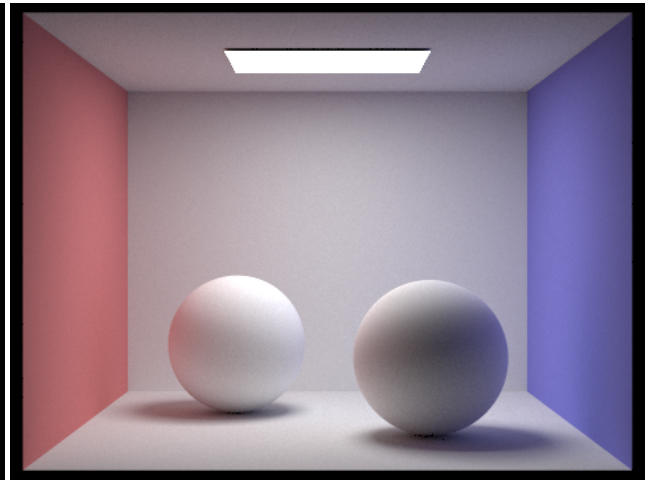
Samples per pixel: 8.



Samples per pixel: 16.



Samples per pixel: 64.



Samples per pixel: 1024.

Part 5: Adaptive Sampling

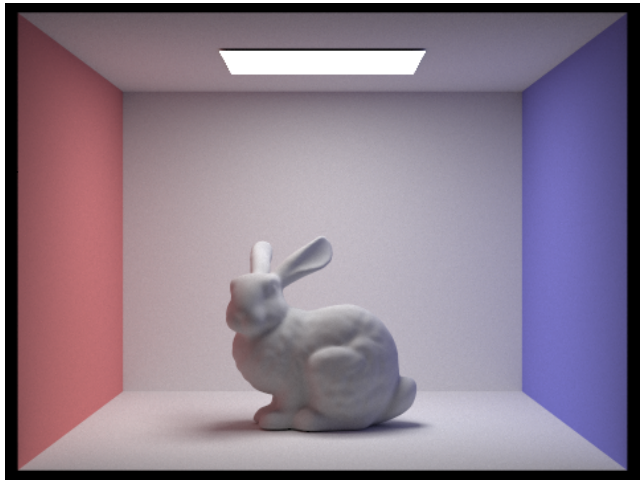
Adapative sampling is a solution we have to eliminating noise while minimizing the amount samples needed, by focusing on increasing the number of samples per pixel for more difficult sections of the image. We try to have pixels that converge faster with low sampling rates use less samples, and other pixels that converge slower use more pixels and reduce noise.

The algorithm given by the project basically checks each pixel whether it has converged or not. We take the mean and standard deviation of the samples of each pixel and measure the convergence using the following formula: I = 1.96 * (stddev / sqrt(n)).

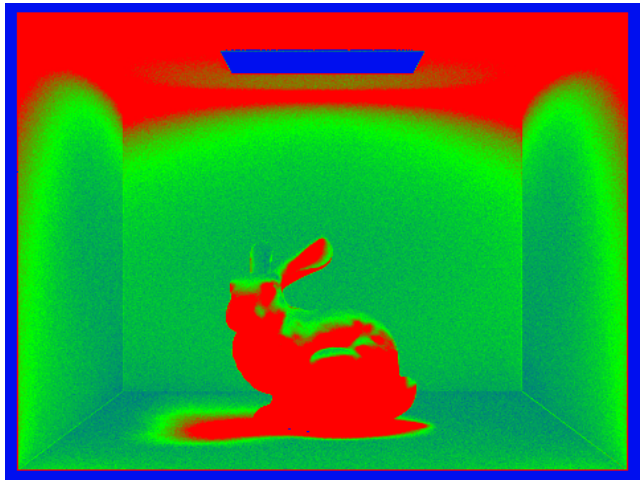
As "I" gets smaller, we can more safely say that the pixel has converged. Once "I" falls under a predetermined "max tolerance" multiplied by the mean of the samples, we then break the loop, effectively reducing the number of samples we have to compute.

To speed up runtime, we also do not check for convergence at each sample. Rather, we have another redetermined parameter which is a "count buffer," which tells us how many samples to wait before doing another convergence check. For example, if the count buffer is 32, we take 32 samples before checking if the pixel has converged. This can drastically reduce the amount of computations we need to do.

Pictured is the bunny image again, rendered with adaptive sampling. I also show the sampling rate image, a kind of "heat map" that shows which pixels take many samples to converge and which don't. The more red the pixel, the more samples it takes before it converges.



The rendered bunny with adaptive sampling.



Our sampling rate "heat map".

And that's the end of Project 3-1 for this class!