CS 184: Computer Graphics and Imaging, Spring 2018

Project 3-1: Pathtracer 1

Larry Yan, CS184

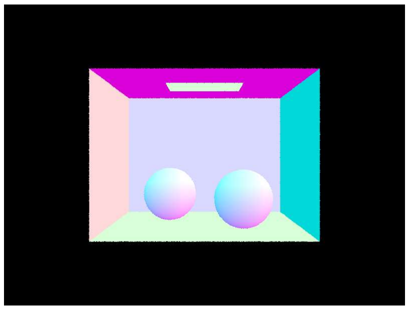
Overview

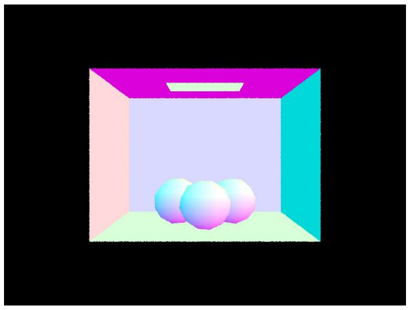
In this project, I implemented elements of a ray-tracing renderer. First, I determined how to set up rays originating from a camera outward to the scene, and sample the illumination at each of those pixel locations. I then implemented the logic for determining if a ray intersected a triangle and a sphere, as well as the locations of those intersections. To speed up these intersections, I set up a bounding volume heirarchy to organize the mesh elements and reduce the number of necessary intersection tests. Afterwards, I implemented lighting, both directly (from light sources to the camera, and bounced off of surfaces from light sources) with hemisphere and light source sampling and indirectly (bounced off of other objects) with monte carlo global illumination. Unfortunately, while I was generally successful with triangle meshes, some element of spherical intersections or normals, or some inverted ray calculation, made indirect lighting on spheres problematic, which I was not able to resolve.

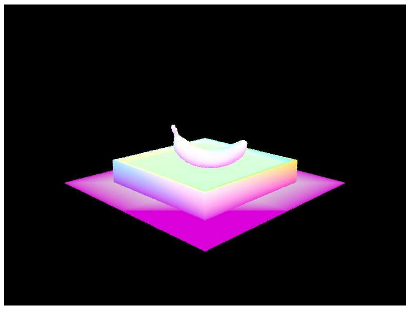
Part 1: Ray Generation and Scene Intersection

For ray generation, in order to generate a sample for a particular pixel, we randomly sample a particular location in that pixel, and convert it into camera space to create a ray (start at camera and go through the corresponding point on the sensor). We then take that ray and transform it into world space with the camera location and rotation matrix to determine what it sees. Additionally, we set the minimum and maximum intersection distances of the ray. We would then take this ray and test it with the primitives, either all of them or through traversing a BVH tree to find the closest intersection, indicating what this sample ray returns. We repeat this process a number of times and average the samples for the pixel value.

For primitive intersections, I used the Moller Trumbore algorithm to determine whether a triangle was intersected, as well as what the barycentric coordinates were. If an intersection did occur between the ray and triangle within the specified bounds, the properties of the intersection were saved into the Intersection object and the ray modified to show that this was the closest intersection thus far. Similarly, for spheres, a modified quadratic was used to determine intersections, and the closest intersection (if any) of the two possible ones was stored if needed.

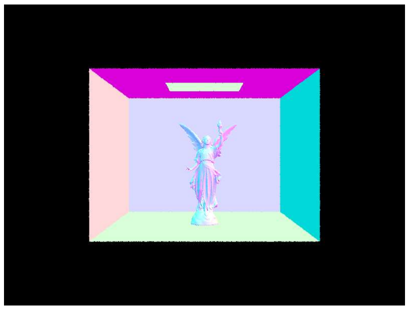

Spheres


Gems

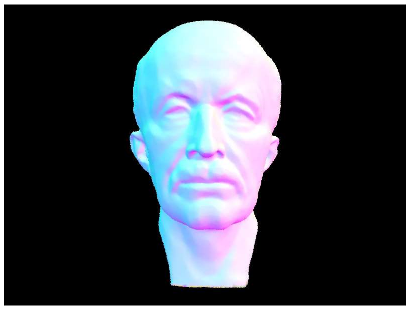

Banana

Part 2: Bounding Volume Hierarchy

To make the BVH, I first took the input list of primitives and checked if they were smaller than the threshold for a BVH node. If it was, then it would be placed in a node and construction would end there. Otherwise, I determined the maximum size axis of the BVH bounding box, and sorted the input primitives based on their centroid locations on that axis, and split them into two equal groups. This would be a relatively simple approach toward separating the primitives into groups that did not require heavy calculations, and would always keep equal numbers on each side of the tree (preventing infinite recursion or unequal splits and producing an even binary tree), even if it did not guarentee perfectly efficient separation. I then called the function on the two new half lists to further split them, until the full BVH was created.


Lucy

Cow

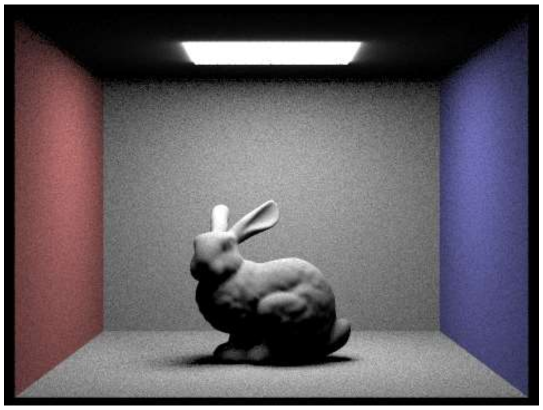

Max Planck

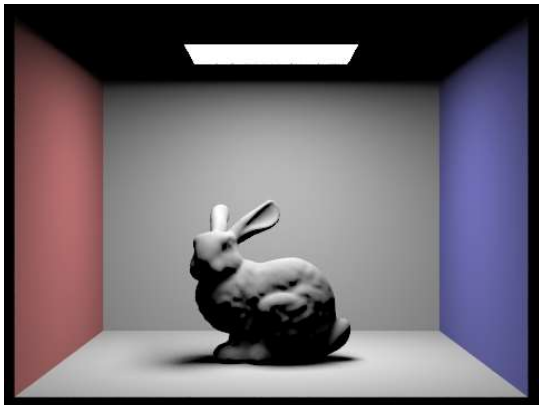
Render times for moderately sized images with BVHs were much faster than without. For banana, it took 3.6152 seconds to render without BVH construction, while with BVH, it took between 0.836 and 1.089 seconds to render. For cow, a more complex structure, render times improved from 18.9347 seconds to between 0.1152 and 0.1918 seconds, an even more dramatic increase. The logarithmic reduction in intersection tests needed makes rendering complex scenes, as well as later lighting tests possible.

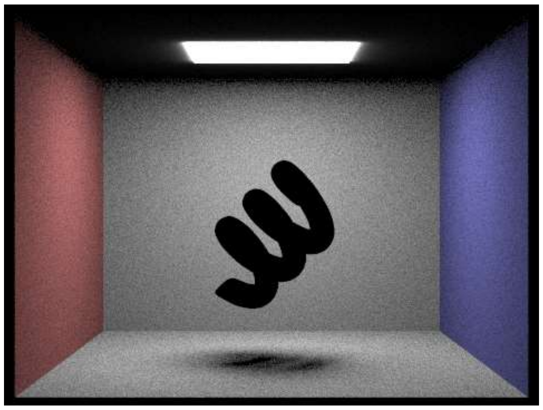
Part 3: Direct Illumination

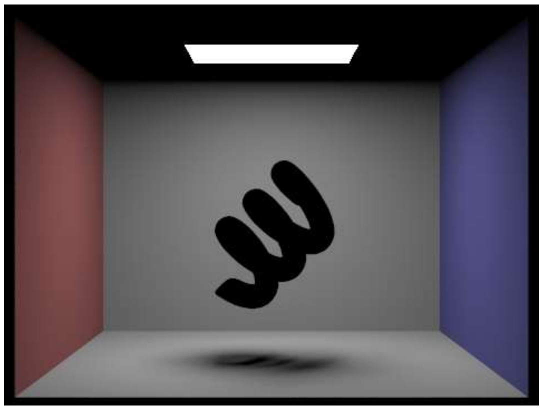
For hemisphere sampling, we sample a random direction from the intersection point on the surface and generate a ray going outward in that direction (first converting that direction into world space). Then, assuming that the ray intersects something, we use the reflection equation to determine the light from the sampled direction to the outgoing direction, based on the product of its reflectance from its bdsf and angles, light emissions, and the cosine of the angle, and normalize by the probability (1/2pi). We then average all of the samples we take to produce the final lighting value.

For importance sampling, instead of sampling in random directions on the hemisphere, we instead loop over all of the lights in the scene, performing ns_area_light samples on each of them instead (or, for point lights, a single sample weighed to be equal to the others). For each of these samples, we sample a ray from the intersection point to a point on the light, and check to see if there are any intermediate obstructions. If there are not, we add the contribution from the light, again weighting it by the bdsf reflectance, returned radiance, and cosine, as well as the returned pdf of the sample. We divide the light totals by ns_area_light to get the total light reflected from that point.

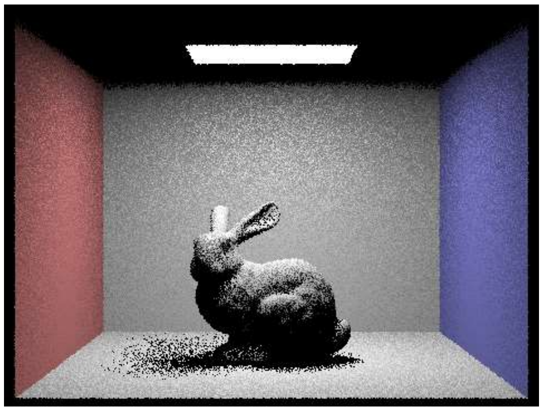

Bunny, hemisphere

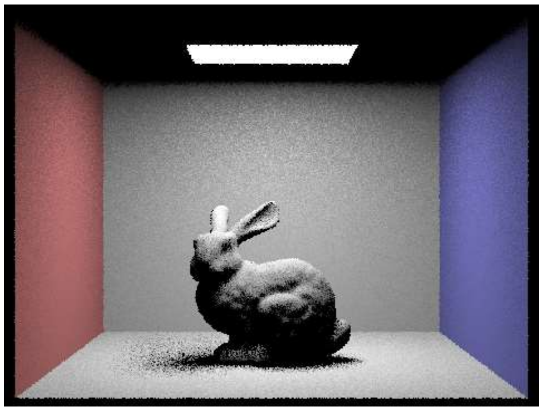

Bunny, light

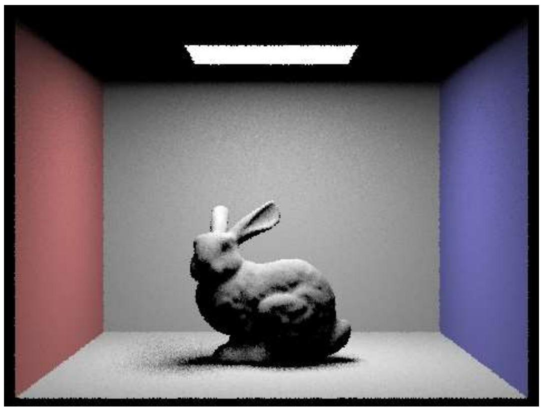

Coil, hemisphere

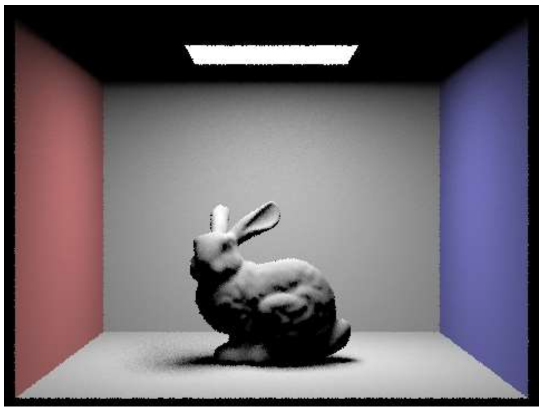

Coil, light

Generally, uniform hemisphere sampling tends to be noisier than lighting sampling, as it is less likely to produce rays that intersect a light source, producing greater variance in illumination between pixels. The walls for the above images tend to be much grainier for the hemisphere sampling, while light sampling is much crisper. Similarly, the shadows are more diffuse for hemisphere, while light has smoother shadows closer to the true amount of light reaching that location.


Bunny, light sampling, 1 light ray


Bunny, light sampling, 4 light rays

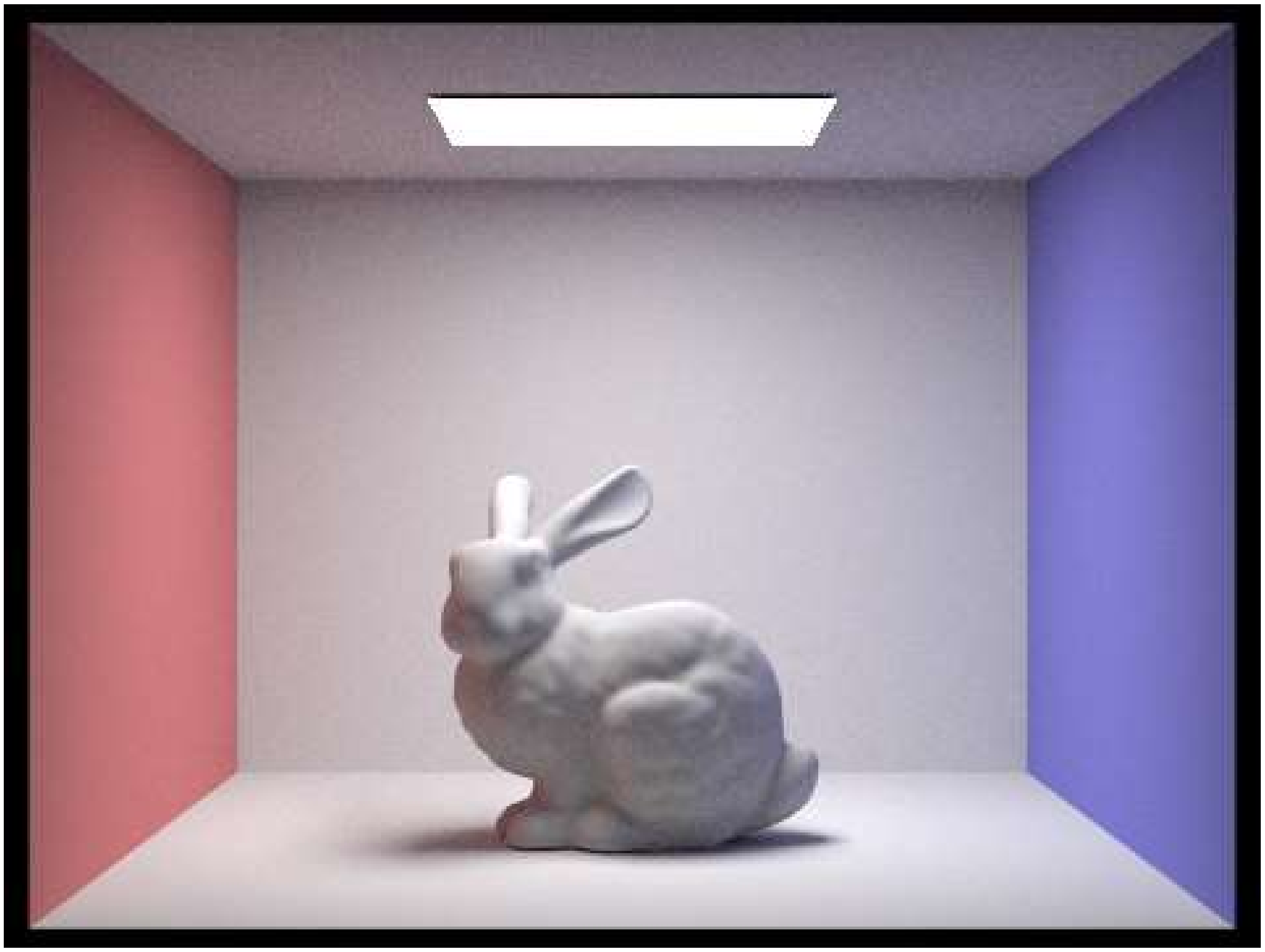

Bunny, light sampling, 16 light rays


Bunny, light sampling, 64 light rays

With fewer samples, there is higher variance in whether the sampled ray for an area light source is occluded, resulting in spotty shadows that are either entirely black or fully lit at 1 sample and grainier shadows at 4 and 16, settling into a smoother transition for 64 samples.

Part 4: Global Illumination

For indirect lighting, instead of just returning light from a single bounce, we additionally with some probability (0.33) generate another random ray from the intersection along a sampled direction, which we recursively call at_least_one_bounce on. We take this light and modify it according to bdsf reflectance, cosine, and pdf, just as we would if this was a light source.


Bunny, 1024 samples.

Part 5: Adaptive Sampling

Link: https://cal-cs184-student.github.io/sp22-project-webpages-yanlarry/