Assignment 3: PathTracer

Billy Chau

Part 1: Ray Generation and Intersection

In order to generate meaningful pixels in the image space, we have to generate rays that represent the lights shooting from our eyes (the camera) to the 3d modeled objects we have created. The transformation from image space to camera space is an important element because it can gives us the tool to represent the captured image in the 3d world and vice versa. Let's take a look at the image below.

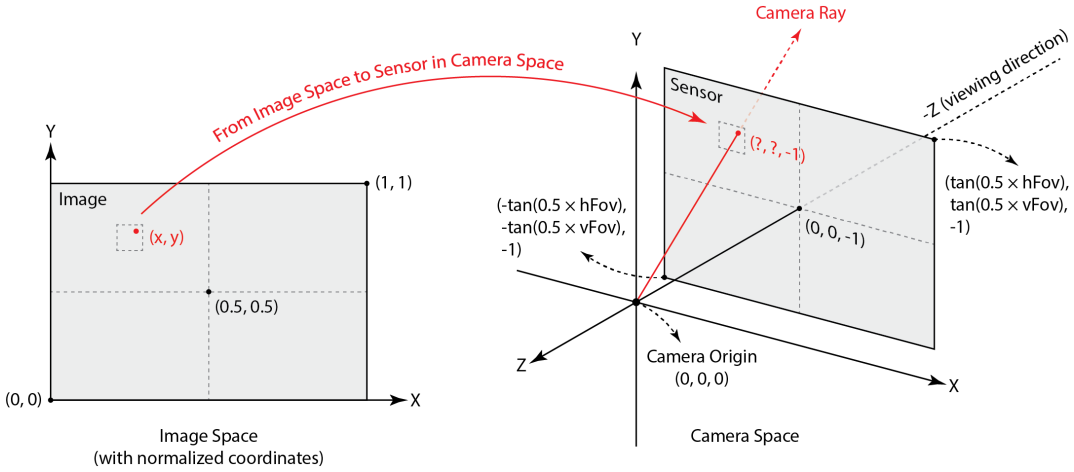


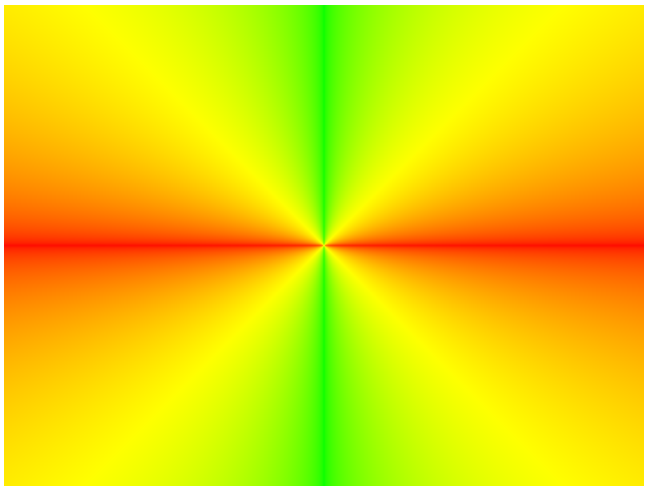
Image to Camera.

Given a ray shooting at pixel location (x,y), we ask for the location of the same pixel in the camera space. In order to calculate an accurate location, we model the camera based on the horizontal and vertical fields of view and assume the image plane lying at the plane z=-1 by convention. Recall that the larger the fields of view, the wider we can see. Using the tangent, we can calculate the width and height of the image plane based on the fields of view. Since the image space is bounded from 0 to 1 both vertically and horizontally, we can represent the location of the pixel by scaling the width and height according to the fields of view. To be specific, scaling the width by 2tan(0.5 x hFov) and the height by 2tan(0.5 x vFov). After that, we have to translate the pixel by (-tan(0.5 x hFov),-tan(0.5 x vFov),-1) such that the origin of the image plane will be at (0,0,-1) just as the image above.

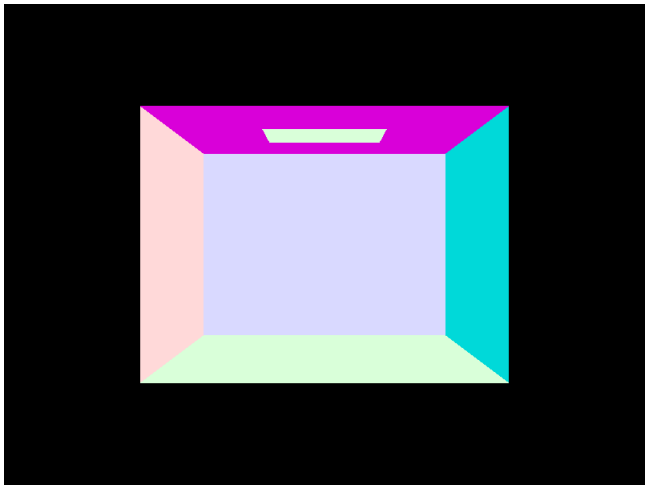
Given the ability to transform a light ray from the image space to the camera space, another transformation is need to connect the ray with the 3d world, which is the transformation from camera space to world space. Luckily, this transformation is given as c2w.

Next, we want to use this generated ray to test whether it has hit any objects in the 3d space, triangle or circle in this case. For triangle, I implemented the moller trumbore intersection algorithm to calculate the intersection point represented by r.o (origin) + t (calculated from moller trumbore) * r.d (ray direction). Using the barycentric constants calculated from moller trumbore, we can check whether the intersection point is within the hit triangle. For circle, I implemented using the implicit function of the circle to solve for t. More specific, solves for the quadratic equation: (r.o + t * r.d - c)^2 - R^2 = 0, where c is the center location of the circle in 3d space and R is the radius of the circle. Here are the results:

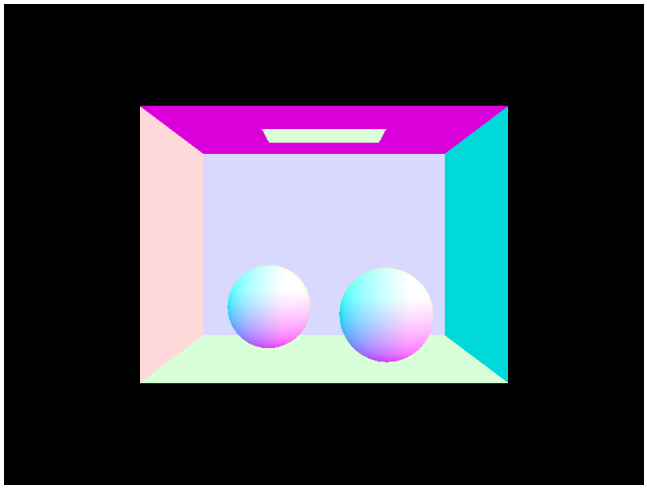
banana.dae



CBempty.dae



CBempty.dae

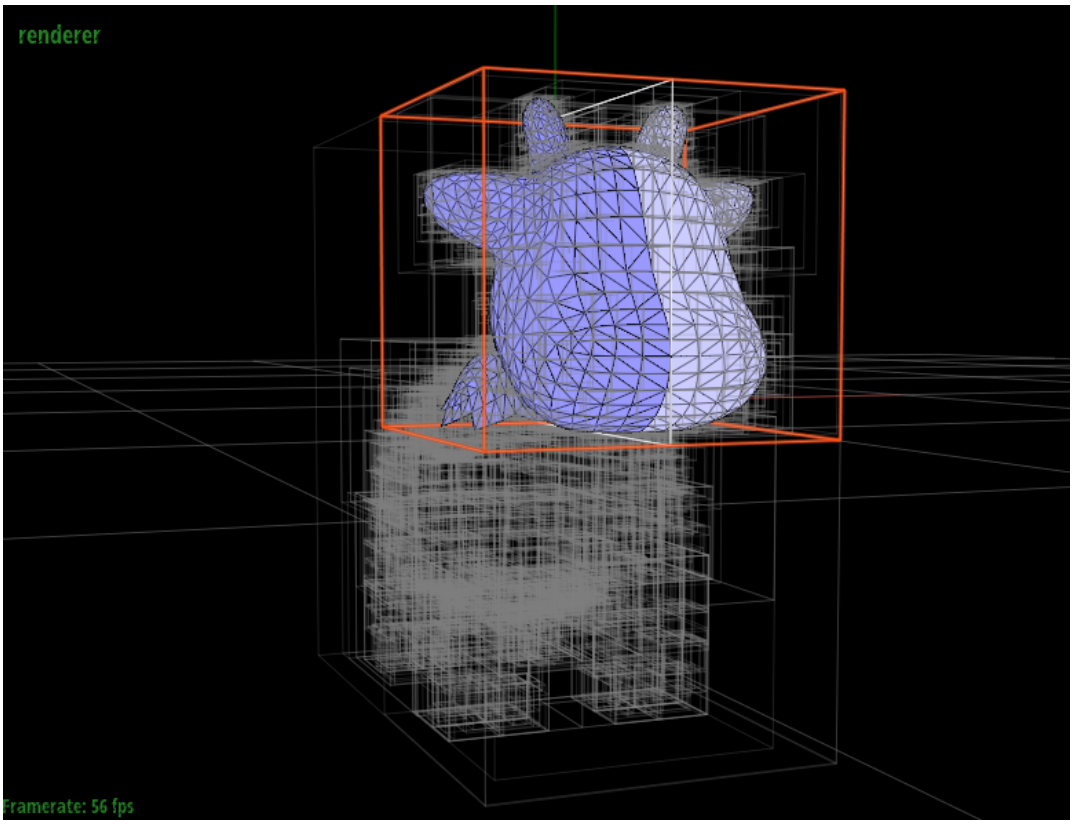


CBspheres_lambertian.dae

Part 2: Bounding Volume Hierarchy

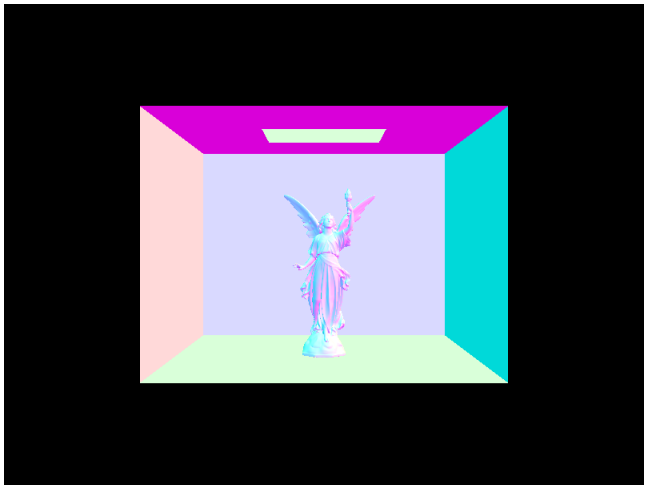
With the ability to generate rays, our next task is to create the bounding volume hierarchy data structure to traverse the objects (triangles) efficiently. Comparing to the brute-force approach, BVH can reduce the time complexity from O(N) to O(logN), where N is number of objects. BVH data structure is a tree-like data structure. All the nodes are BVH nodes in which consist of a bounding box, left and right child BVH nodes, and the objects it represents of. For all internal nodes, there are only bounding box and child nodes while there are only bounding box and objects in the leaf node. In order to construct the whole BVH tree, we have to start with all the objects at the beginning. First, we iterate the objects and build the bounding box from them (passed in as input) for the to-be-created BVH node. Second, we count the number of objects and check whether it is smaller or equal to max_leaf_size (passed in as input). If this is the case, we create the BVH node immediately, assign all the objects to it, and return it. If this is not the case, we use the axis that has the biggest range for splitting.

For the heuristic of picking the splitting point, I chose the bounding box created by the centroid of the bounding box of each object. In this way, I make sure that there is at lease one object at each corner of the box, so it will not run into the case that there is no object in one of the child node. This splitting point is then used as the decision criteria to put the objects into either left or right child node of this newly created internal node. Since the base case is that every BVH node has at most max_leaf_size objects, we have to recursively call this function to populate the correct data into its child nodes until the base case is met.

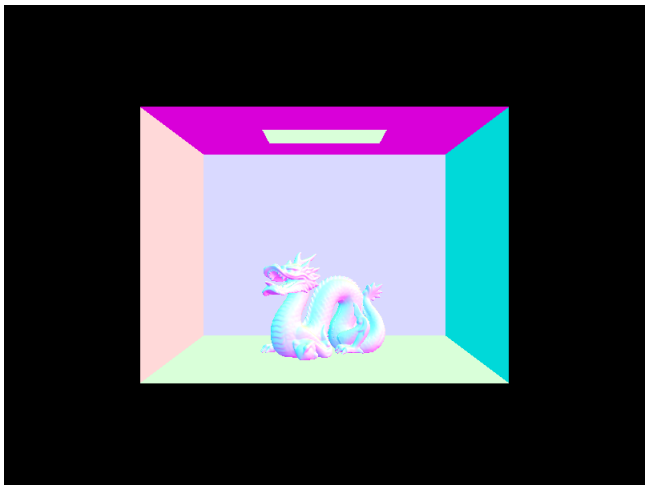


Iterating the bounding box of the BVH nodes.

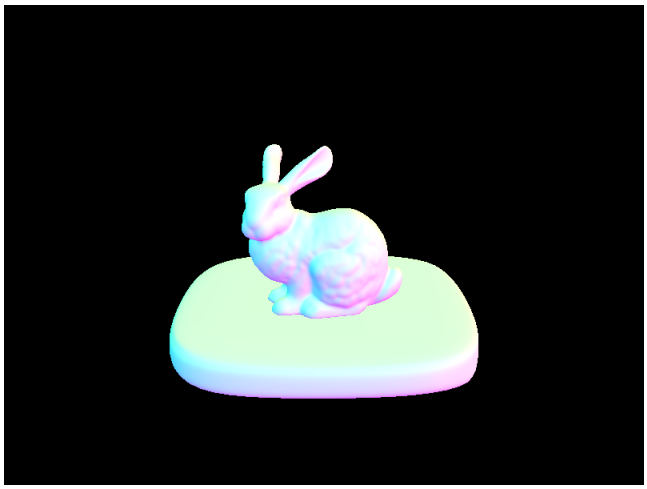
maxplanck.dae



CBlucy.dae



dragon.dae



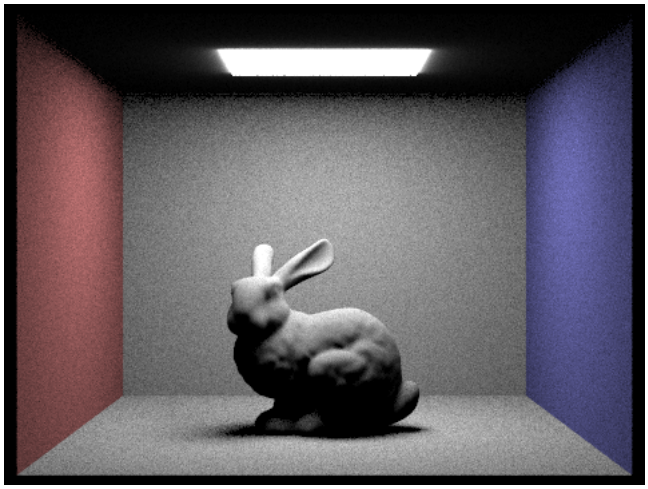
CBbunny.dae

With the implemented BVH acceleration, the performance is significantly improved. For example, rendering CBlucy.dae and dragon.dae could take up to few minutes if rendered using brute-force approach whereas they are rendered within a second or two with the BVH acceleration. It is obvious that the performance improvement is due to the O(logN) nature of the tree traversal and the balance of the tree.

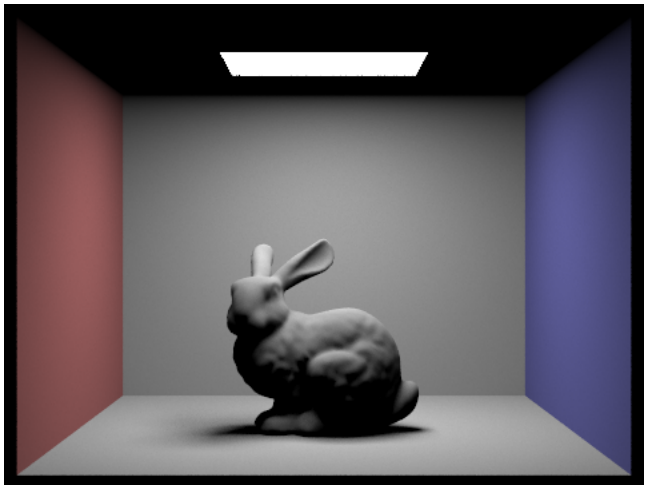
Part 3: Direct Illumination

Current result does not have realistic color because it is doing local lighting rather than raytracing. There are two basic lighting captured in raytracing: direct and indirect lights. Direct light is simply the light directly hitting from the light source to the image plane. Indirect light is all other lighting effect due to the bouncing lights from the direct light hitting off objects. In this part, we implemented two approaches for direct lighting on a point: hemisphere and importance sampling. Hemisphere sampling is a monte carlo sampling over a hemisphere. Since monte carlo sampling is unbiased, the normalized light eventually converges to the expected value given enough samples of light rays. Given a ray hitting an intersection in the direction of d, we want to sample a new ray over a hemisphere going off from the intersection point. On the other hand, importance sampling is sampling at the ranges of direction where the new ray directs to the light source (luckily the sampling function is implemented for us by the function sample_L). And both approaches require a pdf for monte carlo sampling. Since we only care about the light if the new ray hits a light source in this part hence the name of direct lighting, we simply ignore other directions if it does not hit any light source. To be more precise, for each sample, we first sample a new ray (depends on hemisphere or importance sampling) from the hit point; if the new ray hits a light source, we get the bsdf of the intersected material and calculate the dot product between the new ray and the outgoing ray, which is going to the camera. After getting all these information, we can perform our monte carlo sampling by aggregating the value of each sample calculated by bsdf * radiance * dot product / pdf. Finally, we need to normalize the result by the number of samples.

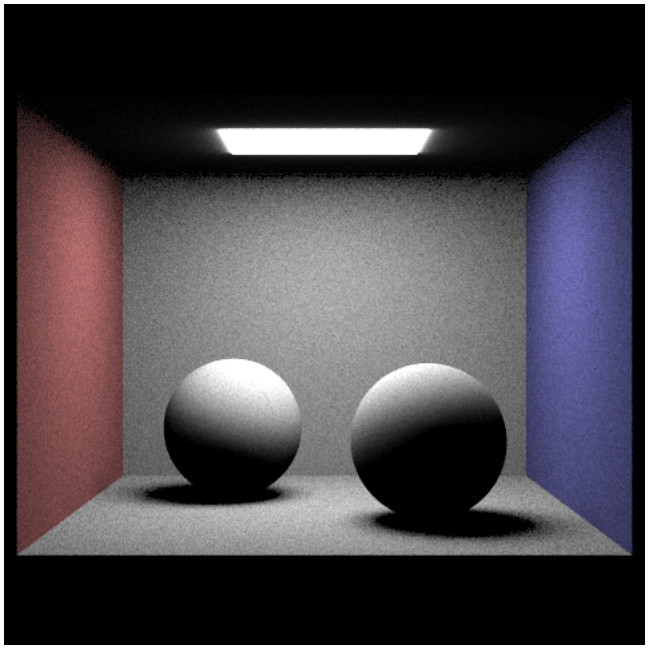
Zero Bounce Light.



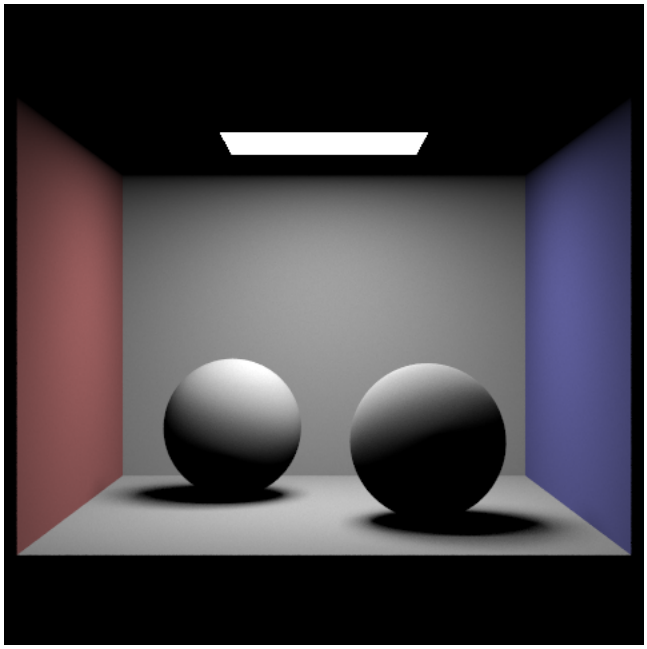
CBbunny.dae with hemisphere sampling.



CBbunny.dae with importance sampling.



CBspheres_lambertian.dae with hemisphere sampling.



CBspheres_lambertian.dae with importance sampling.

As we can see from above, hemisphere sampling tends to produce a lot of noise while importance sampling does not given the same sampling configurations. The noise in hemisphere sampling mainly comes from the useless sampled directions. Importance sampling does not have noise because it does not sample any directions that are not facing the light source, thus minimizing the randomness.

dragon.dae with light rays = 1.

dragon.dae with light rays = 4.

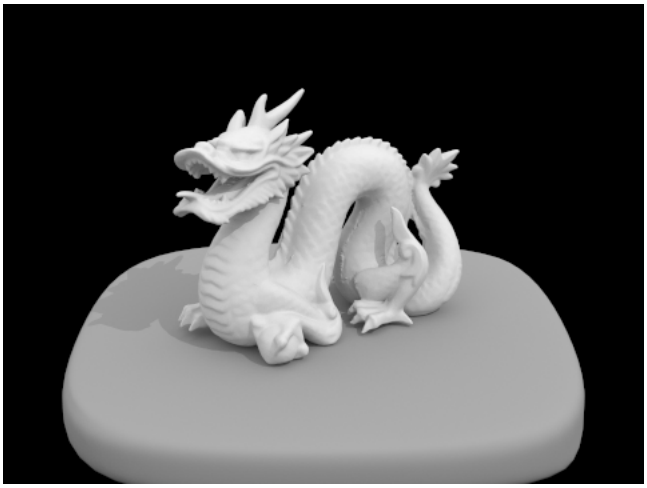
dragon.dae with light rays = 16.

dragon.dae with light rays = 64.

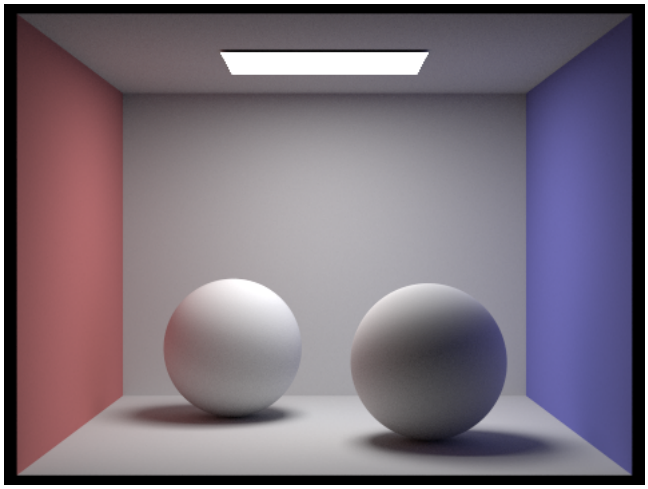
Given sample per pixel = 1, we can conclude that increasing the number of light rays can improve the direct lighting significantly. The time importance sampling takes to converge is faster than hemisphere sampling because it does not spend time on useless directions.

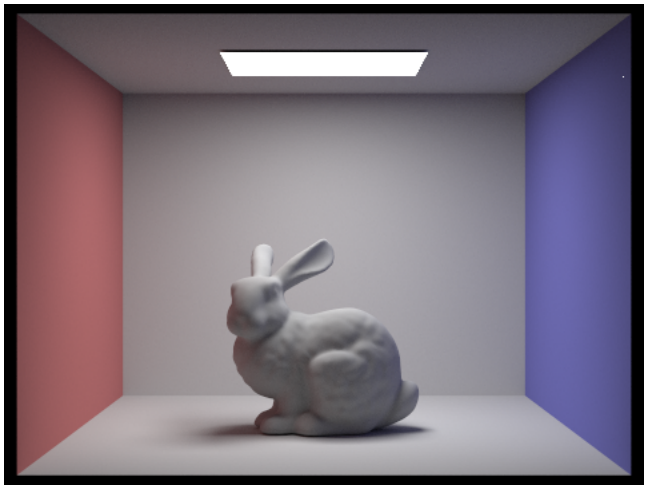
Part 4: Global Illumination

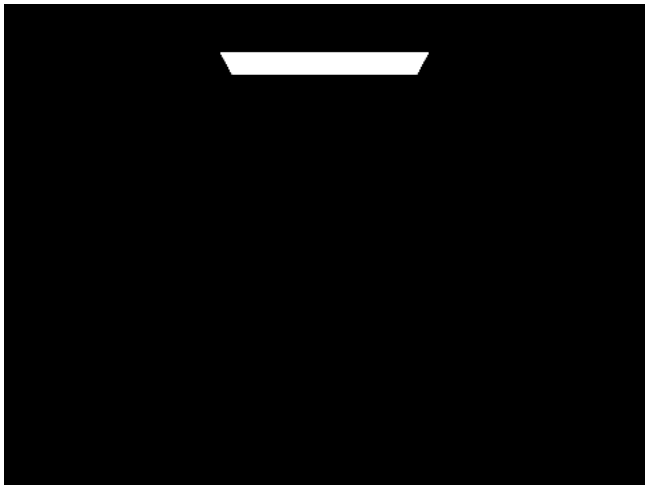
After implementing direct lighting, it is important to implement the indirect lighting part for realistic lighting effect. In other words, we only care about the lights that are reflecting from other objects rather than light sources. Since such lighting could come from any objects and those objects could have lights coming from infinite objects (self or others), there is a possibility of infinite recursion. Therefore, we have to add the probability of the Russian Roulette terms to end the infinite recursion. Specifically, if a random sample from the interval [0,1] is smaller or equal to the suggested terminating probability (0.6 or 0.7), we terminate the recursion. Thank you Monte Carlo again, the expectation with the terminating probability will also converge to the realistic light given enough samples. The implementation of global illumination is very similar to the direct illumination except the terminating probability and the bsdf sampling. Using the similar equation from the previous part, we can perform our monte carlo sampling by aggregating the value of each sample calculated by (bsdf * radiance * dot product) / (pdf * terminate_pdf). It is important to note that the bsdf in this case is not deterministic because of the face that we don't care about direct light and the bsdf depends heavily on the material (its distribution). Finally, it is crucial to note that we are not collecting the result from multiple sample light rays because the result of the indirect lighting does not depend on the light source anymore.

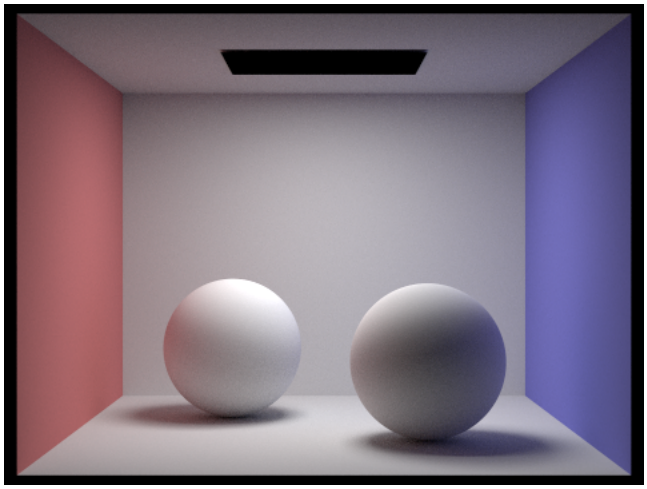


dragon.dae with global illumination.

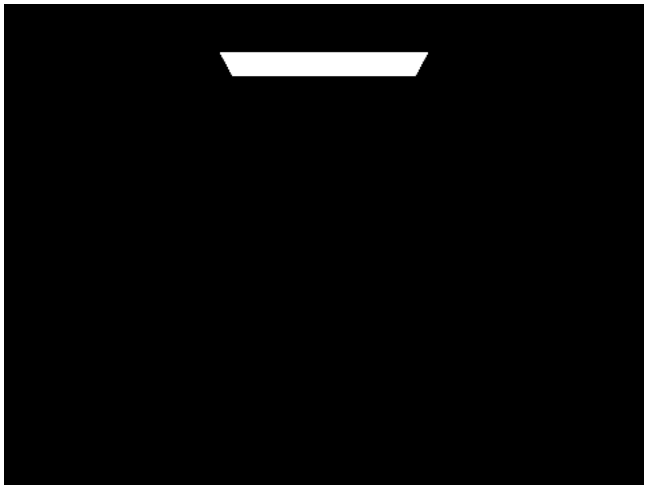

CBspheres_lambertian.dae with global illumination.

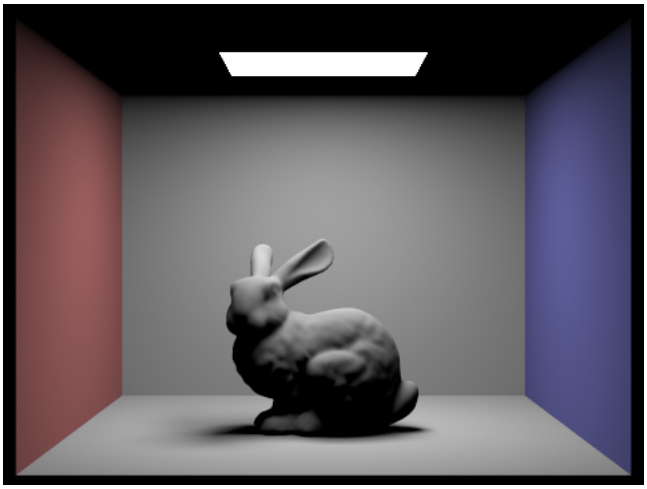

CBbunny.dae with global illumination.

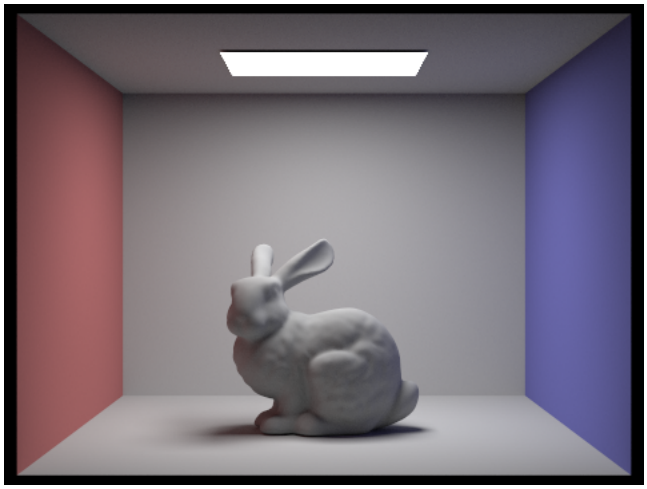

CBspheres_lambertian.dae with only direct illumination.


CBspheres_lambertian.dae with only indirect illumination.

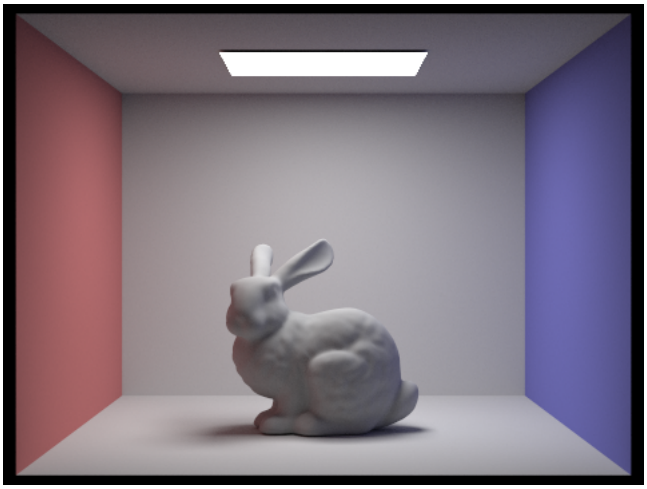
As we can see from above, the direct illumination only captures the light source while the indirect illumination captures all bouncing lights except the light source.


CBbunny.dae with max_ray_depth = 0 and 1024 samples per pixel.

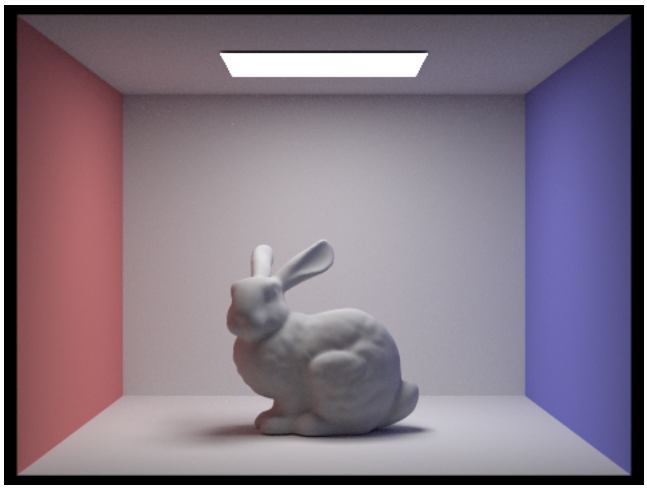

CBbunny.dae with max_ray_depth = 1 and 1024 samples per pixel.



CBbunny.dae with max_ray_depth = 2 and 1024 samples per pixel.

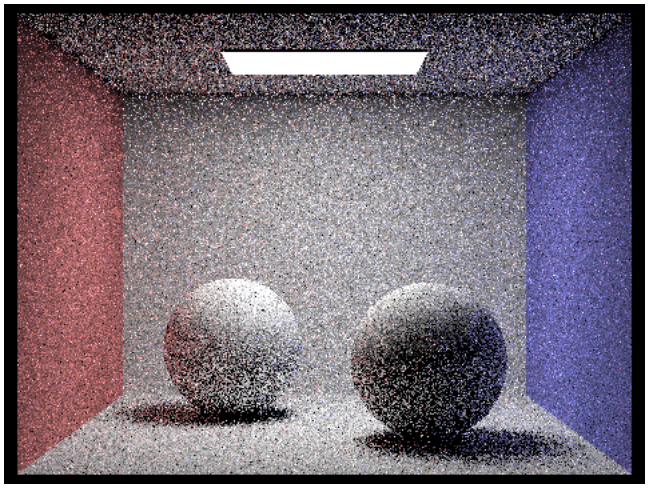


CBbunny.dae with max_ray_depth = 3 and 1024 samples per pixel.

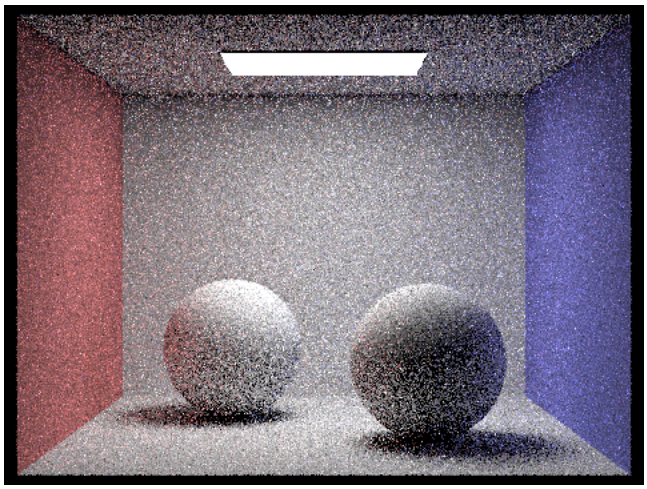


CBbunny.dae with max_ray_depth = 100 and 1024 samples per pixel.

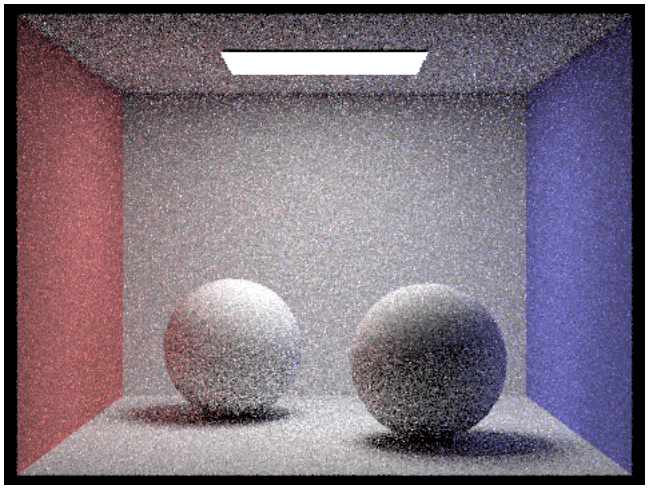
As we can see from above, the image gets brighter when the max_ray_depth increases. This is expected because the higher the max_ray_depth is, the deeper the bounced light is, thus aggregating into a more realistic image.



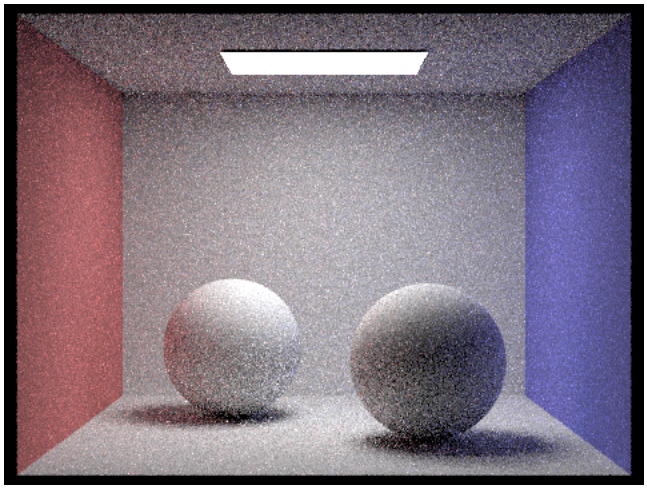
CBspheres_lambertian.dae with light rays = 4 and 1 samples per pixel.



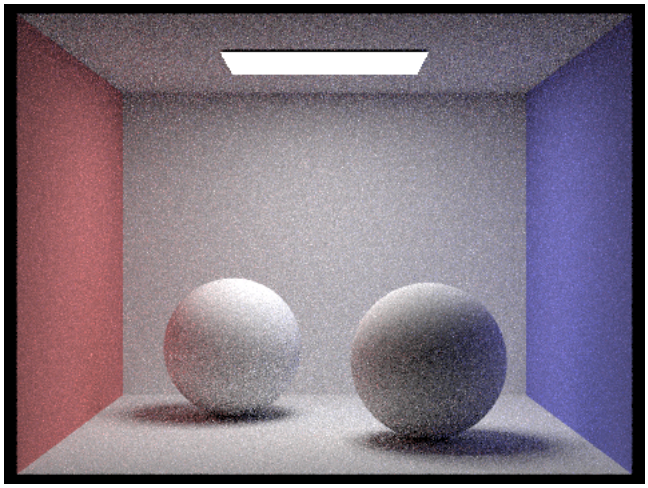
CBspheres_lambertian.dae with light rays = 4 and 2 samples per pixel.



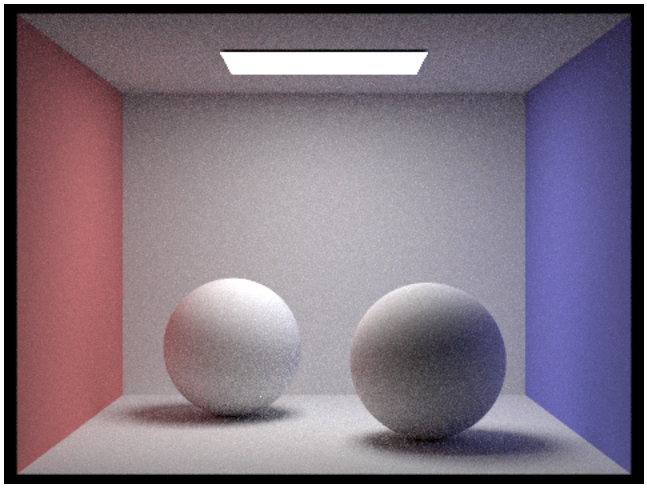
CBspheres_lambertian.dae with light rays = 4 and 4 samples per pixel.



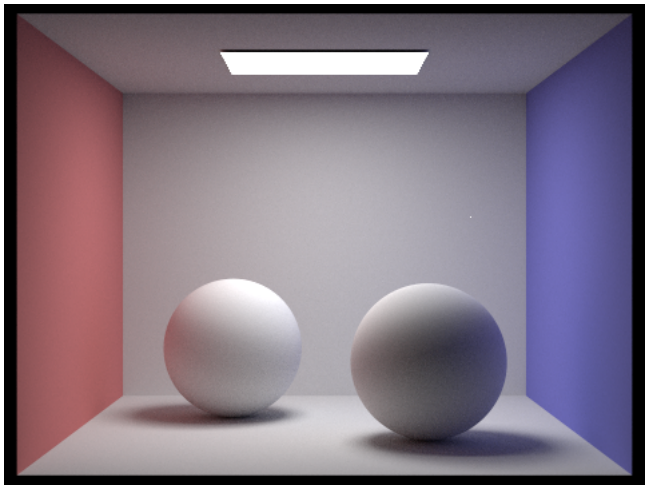
CBspheres_lambertian.dae with light rays = 4 and 8 samples per pixel.



CBspheres_lambertian.dae with light rays = 4 and 16 samples per pixel.



CBspheres_lambertian.dae with light rays = 4 and 64 samples per pixel.

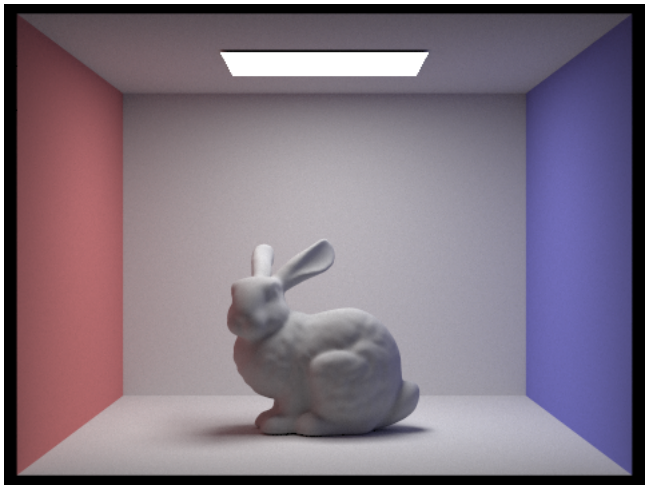


CBspheres_lambertian.dae with light rays = 4 and 1024 samples per pixel.

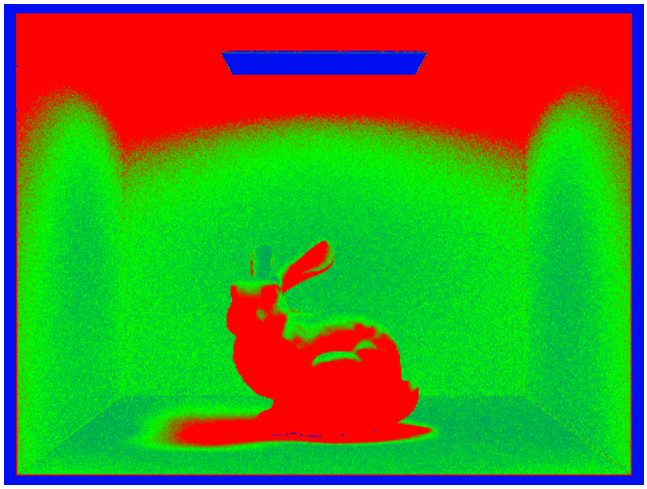
As we can see from above, the image eventually gets higher quality as the number of samples per pixel increases. Unfortunately, there is still slight noise with 1024 samples per pixel because it is all about the waiting game if we want higher quality result.

Part 5: Adaptive Sampling

If this is all about the waiting game, is there anything we can improve upon? Yes, we can avoid further raytracing at certain pixels when their colors have converged. We can use the z-test to check the convergence of illuminance of each pixel. Specifically, we can calculate the mean and the variance on the fly while sampling. We can then use them to construct 95% confidence bound such that +/- 1.96 * std / sqrt(N) from the mean is within the confidence bound. Since we want to ensure the convergence of the illuminance, we are looking for a narrow bell curve. Therefore, we have to check if the following condition has been met: 1.96 * std / sqrt(N) <= 0.05 * mean. If the following condition is met, we stop the raytracing at that pixel. Although z-test is a good approximation, I think student t-test could be a better convergence test because it can handle small number of samples.



CBbunny.dae.



CBbunny.dae rate graph.

Looking at the image and rate graph, we can conclude that the rate graph matches the result of the image. First, the light is sampled the least times because there is indirect lights hitting. Second, the bunny's body is sampled the most times because there are a lot of shadows and shade created by the multiple bounces indirect lights. Finally, the walls next to the bunny are sampled not as often as the bunny's body but also not as rare as the light because it mostly accounts for the few bounces indirect lights coming off from the ground and other walls.