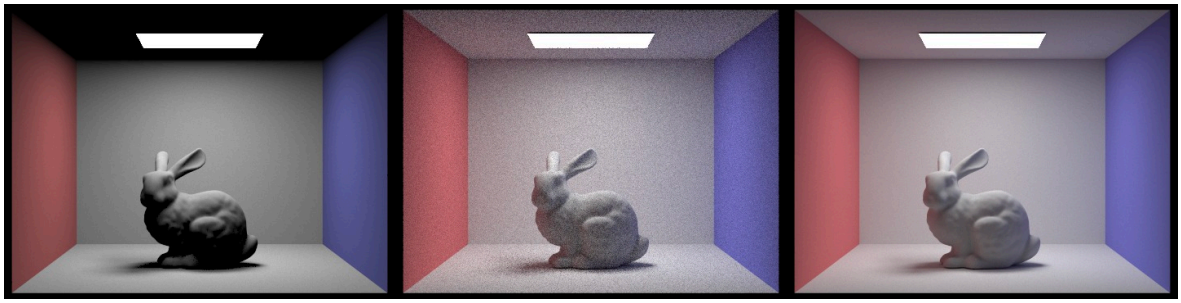
CS 184: Computer Graphics and Imaging, Spring 2024

Path Tracer

Ethan Tam



Overview

In this Path Tracer assignment, I went on a journey in building a renderer capable of creating real looking images through the use of the path tracing algorithm. To create this path tracer, I first created a foundation of important functionalities like ray generation, scene intersection, and shading computations. Throughout this assignment, I encountered many delays in rendering especially when I'm rendering large and detailed .dae files. This in return prompted me to utilize Bounding Volume Hierarchy (BVH), to optimize the function of recursively partitioning primitives into left and right groups of nodes. My approach involves sorting the primitives based on their centroids and if it is above or below the average. In addition, I added direct illumination and global illumination to my renderer. Direct illumination uses Bidirectional Scattering Distribution Function (BSDF) and handles zero-bounce illumination. As for global illumination I dove deeper into the implementation of the recursive technique to estimate light interaction and also implemented Russian Roulette for unbiased randomness.

Looking back at the assignment, the journey was rough trying to understand and conceptualize the ideas of rendering complexities in 3D projects. There were many times where I had to run renders for a good chunk of time just for it to error or crash, but this is just a reminder of how inherently difficult rendering is to implement and understand. Even with these challenges, I was able to debug and complete the assignment which brought me a lot of satisfaction and lessons I've learned. Through the process, I also deepened my understanding of principles in ray tracing, gaining insight into the complexity of rendering engines.

Part 1: Ray Generation and Scene Intersection (20 Points)

Walk through the ray generation and primitive intersection parts of the rendering pipeline.

To implement ray generation, I first transform the image coordinates to camera space and then generate the ray in camera space. Afterwards, I transformed the resulting ray into world space. To do the first step, I drew out the image coordinates and how it will translate to camera space and then I converted hFov and vFov to radian forms. Because we are given the origin of the camera we can calculate the camera direction, we can use 2*tan(0.5*hFov) and 2*tan(0.5*vFov) to help solve for the x and y direction. To calculate the x direction, I subtracted the x by .5 and the multiply it by the width (2*tan(0.5*hFov)). To calculate the y direction, I subtracted the y by .5 and the multiply it by the width (2*tan(0.5*vFov)). With the camera directions I can generate the camera space's ray. I then also set the min_t to nClip and max_t to fClip. The final step is to transform the resulting ray into world space. I did that by multiplying the direction of the ray by the rotation matrix (camera to world). I then normalized the resulting direction.

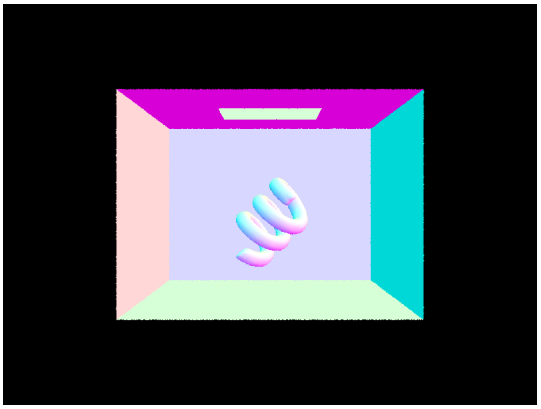
For the triangle intersection algorithm I implemented in this part of the project, I implemented raytrace_pixel(...). When I implemented raytrace_pixel in part 1, I created a loop and went through all the samples while keeping track of the total Vector3D for later uses. In the iteration, I access a random Vector2D by using gridSampler->get_sample(). Then I add the Vector2D values indexed at the 0 index and 1 index to pixel coordinates with x and y. Then I normalized the values by dividing it by sampleBuffer.w and sampleBuffer.h. Afterwards, I used the updated x and y value coordinates to create a ray by using camera->generate_ray() again. With the result, I passed it into est_radiance_global_illumination. At the end of the iteration, I added the resulting Vector3D to the total and repeated

with the rest of the loop iterations. Finally, I divide Vector3D total by our total number of samples which is the same value as ns_aa and update the sampleBuffer at the final Vector3D.

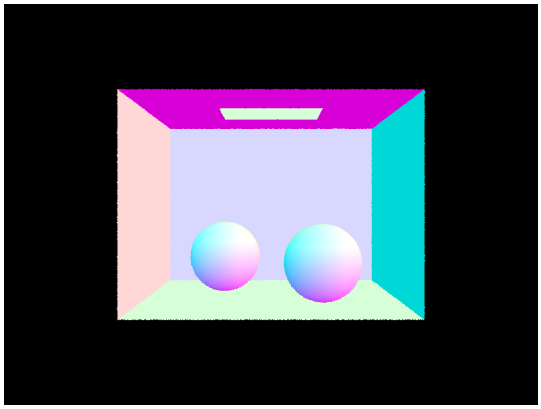
Explain the triangle intersection algorithm you implemented in your own words.

By using the algorithm, Möller-Trumbore Algorithm and barycentric coordinates, I was able to implement ray triangle intersection functionality. For the function has_intersection, I calculated the u, v, and t values, by applying the Möller-Trumbore Algorithm which uses the triangle's VEctor3D points, the point's normals, and the ray's direction and origin. After the calculations, I was able to get the t value, beta (u) value, and gamma (v) value. I also did a three line test to make sure that all the barycentric coordinates are bigger than or equal to 0. I then checked to make sure the calculated t value is in the range of the ray's min_t and max_t parameters. If it is then I want to update the max_t value of the ray to the value of t and return true. Now in the intersect method, I also used the Möller-Trumbore Algorithm to get the barycentric coordinates as well as the vector. I then used the three line test to see if the t value is in range of the r's min and max. If it is, we want to update the values of the intersection structure. In addition, I computed the surface normal and set it to isect->n. I then normalized it and updated the other pertinent attributes like the primitive pointer and intersection's BDSF. Finally, I updated the ray's max_t value to the t value.

Show images with normal shading for a few small .dae files.



CBcoil.dae



CBspheres_lambertian.dae

cow.dae

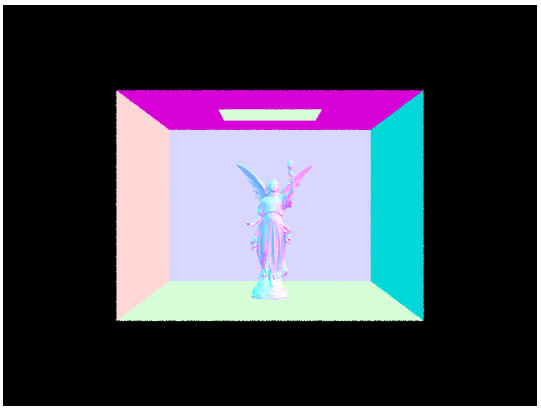
Part 2: Bounding Volume Hierarchy (20 Points)

Walk through your BVH construction algorithm. Explain the heuristic you chose for picking the splitting point.

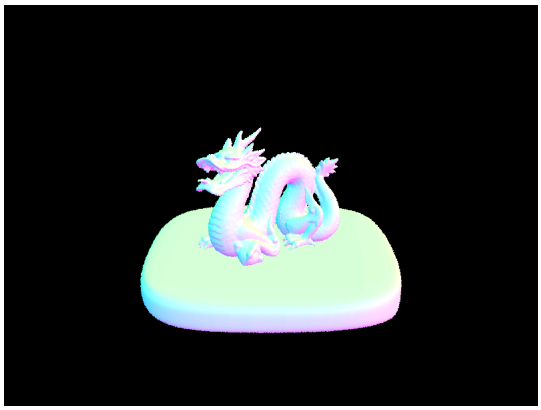
For the BVH construction algorithm, I started by declaring a bounding box variable and then used a for loop to go from the start to end and calculated the bounding box of a list of primitives. In each iteration, the for loop accesses the primitive pointer and then immediately dereferences it. Then it gets its respective bounding box by using the get_bbox() function. With the bounding box that I created outside of the for loop, I can use the function expand() with the bounding box of the current primitive pointer passed into it. The heuristic that I implemented in this part of the homework for picking my splitting points is the average of the centroids that is on the axis with the biggest bounding box extent. My next step is creating a new BVHNode with the bounding box I just created. By using the distance function from start to end, I'm able to keep track of the total number of primitives in the whole entire for loop. I'm also able to keep track of the total sum of all centroids in each primitive's bounding box. Then I calculated the average of all the centroids in each axis. Afterwards, I look to see if the primitives' total size that the for loop iterate over is less than or equal to max_leaf_size. If it is, we want to set node->start equal to start and node->end equal to end. I then returned the node. This "less than or equal to

max_leaf_size" conditional statement will be a base case for the recursive implementation that I used. Now if the primitives' total size is >max_leaf_size, we want to divide the primitives to two groups, a left and a right. Then I retrieved the axes with the biggest bounding box extent. We know the node is not a leaf node and that there is more work to be done. We need to divide the primitives into left and right groups. We first obtain the axis (x, y, or z) that has the greatest bounding box extent. Afterwards, I split and sort the primitives in the groups using std::partition() where I passed in a start, end, and some sort of comparator. For this comparator, I used a lambda function that takes the pointer to the current primitive, a pointer to the largest axis, and the calculated centroid average. Looking at the current primitive's bounding box, we can then see if the centroid at the axis is less than the centroid average. If it is, we want to put the primitive to the left group. If it is not, we want to put the primitive on the right group. The std::partition() function will return a new iterator with the last primitive at the end stored in the left node. Because of this, I can create our recursive statement by setting the left and right values of the node to the result of construct_bvh. The left value of the node will be set to the result of construct_bvh(..) with the start iterator, the new iterator (the middle), and the max_leaf_size. The right value of the node will be set equal to the result of construct_bvh(..) called on the new iterator (the middle), the end iterator, and the max_leaf_size.

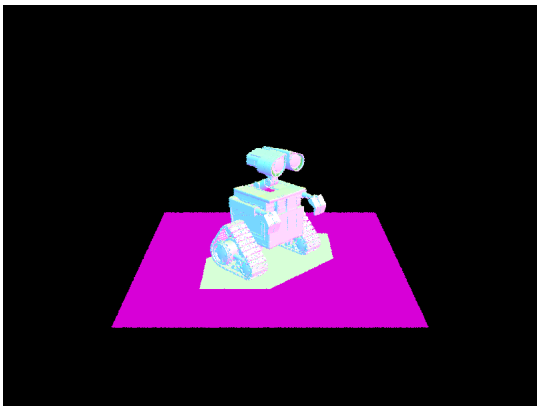
Show images with normal shading for a few large .dae files that you can only render with BVH acceleration.



CBlucy.dae



dragon.dae



wall-e.dae

Compare rendering times on a few scenes with moderately complex geometries with and without BVH acceleration. Present your results in a one-paragraph analysis.

Using BVH acceleration, the rendering times are exponentially quicker than before. I tested 3 dae files. The first one is CBlucy.dae which took 63.9386 seconds without BVH acceleration and .0413 seconds with BVH acceleration. I then tested the wall-e.dae file and it took 126.9798 seconds without BVH acceleration and .05845 seconds with BVH acceleration. Lastly, I tested the dragon.dae file which took 41.6483 seconds without BVH acceleration and .05595 seconds with BVH acceleration. I also noticed how the number of intersection tests per ray reduced with BVH acceleration and this is probably because BVH acceleration reduces the number of rays checked by using bounding boxes. By checking if rays that hit primitive in a bounding box, we can reduce the amount of interactions tests on rays. By using BVH acceleration we reduced the complexity from O(N) to O(log(N)).

Part 3: Direct Illumination (20 Points)

Walk through both implementations of the direct lighting function.

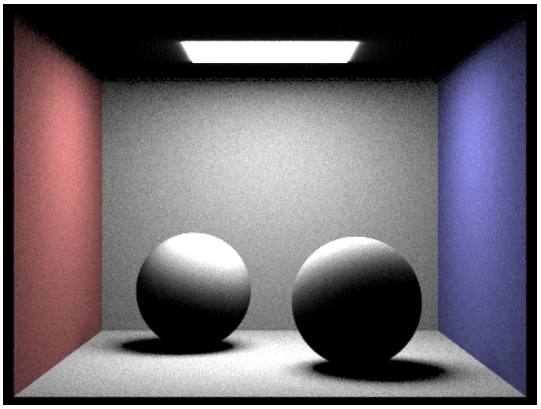
To implement direct lighting with uniform hemisphere sampling, the first step I took was to create a for loop which iterates num_samples of times, in order to compute the Monte Carlo estimator. In each iteration, I uniformly sample the group of incoming ray directions using CosineWeightedHemisphereSampler3D() and get_sample(). Next, I created a new ray that originates from the sampled

direction's hit_p. In addition, I set the min_t of the new ray to EPS_F in order to prevent any calculation and precision issues. Next, invoked bvh->intersect with the new ray passed in and &iter to check if there is a intersection and light source that is valid. If it is true, we want to update the intersection. I then calculated the response function of the Monte Carlo estimator by using isect.bsf->f() and passing in w_out and the sample I got from CosineWeightedHemisphereSampler3D().get_sample(). The next two steps I took was calculating the emitted light from the new intersection by calling bsdf->get_emission() on the intersection and calculating the cosine theta on our sample. I then multiplied the emitted light vector, the response function, and the calculated cosine theta. Finishing it off, I normalized the product with the PDF (1.0 / (2.0 * PI). This iteration process is repeated num_samples amount of times and then after the for loop, I normalized it one more time by the number of samples (num_samples). I then returned the result.

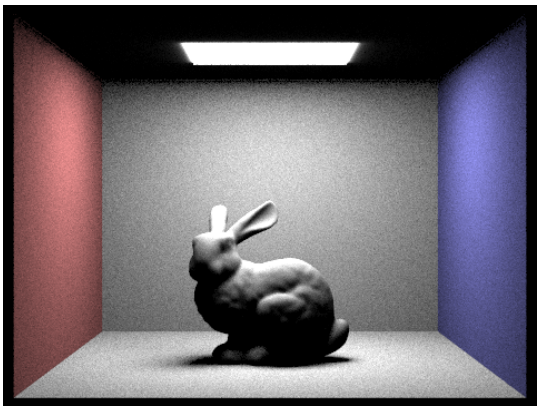
To implement direct lighting with lighting importance, I started by creating a foreach loop to go through each Scene Light pointer that is in scene->lights. Then I did is_delta_light() on every scene light pointer. If is_delta_light() is true, I sample once because all the samples are the same if it comes from the same point light source. I obtained the emitted radiance by using the current light's sample by invoking sample_L() with hit_p, the distance to the light variable, a pdf variable, and address of w. The variable of the distance, pdf, and were declared beforehand. I then created a new ray with hit_p as the origin and w as the direction. I then calculated the ray's min_t and max_t with precision errors considered with EPS_F. Using bvh->has_intersection, I check if there is no intersection between the new ray and the object in the scene. If there is no intersection, we know that there is nothing between the hit point and the light source, hence we can apply the same calculation in task 3 and then normalize the result with the PDF. The result of that will then be added to L_out. However, if the light isn't considered to be a point light source, we just sample ns_area_light amount of times. The same process as point light sources is then repeated num_samples amount of times in a loop. After the loop, I normalized the result by num_samples. This whole procedure is repeated each time with all the scene lights. After that L_out is returned.

Show some images rendered with both implementations of the direct lighting function.

Uniform Hemisphere Sampling

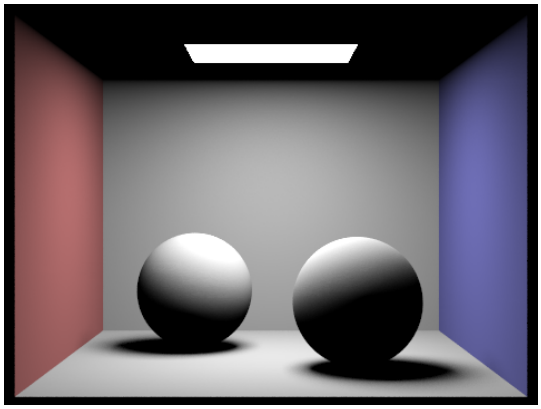


CBspheres_lambertian.dae

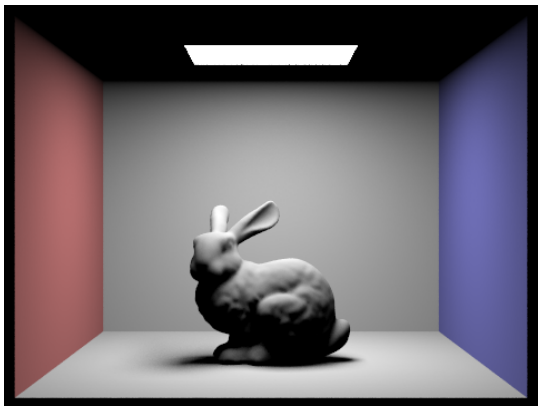


CBbunny.dae

Light Sampling

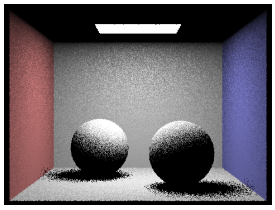


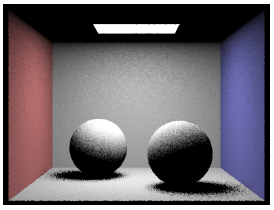
CBspheres_lambertian.dae

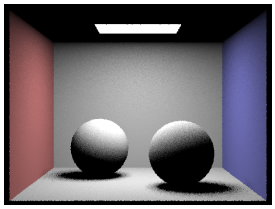


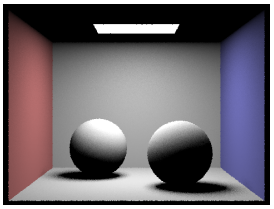
CBbunny.dae

Focus on one particular scene with at least one area light and compare the noise levels in soft shadows when rendering with 1, 4, 16, and 64 light rays (the -l flag) and with 1 sample per pixel (the -s flag) using light sampling, not uniform hemisphere sampling.


1 Light Ray - CBspheres_lambertian.dae


4 Light Rays - CBspheres_lambertian.dae


16 Light Rays - CBspheres_lambertian.dae


64 Light Rays - CBspheres_lambertian.dae

Compare the results between uniform hemisphere sampling and lighting sampling in a one-paragraph analysis.

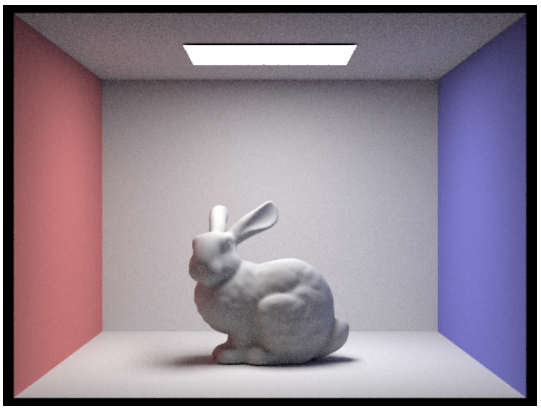
When it comes to Uniform hemisphere sampling it seems to make noisier results compared to the results of direct lighting sampling. This result is because uniform hemisphere sampling selects rays at random making some rays intersect with objects and other rays not. This creates a grainy and dark noisy pixelated image. In addition, the scenes with point light sources are challenging for uniform hemisphere sampling because they have a very small probability of hitting tiny light sources. On the other hand, importance sampling samples from a light source makes it have less noise in its images because it makes sure all samples will positively add to the image. This results in smoother and less rough images.

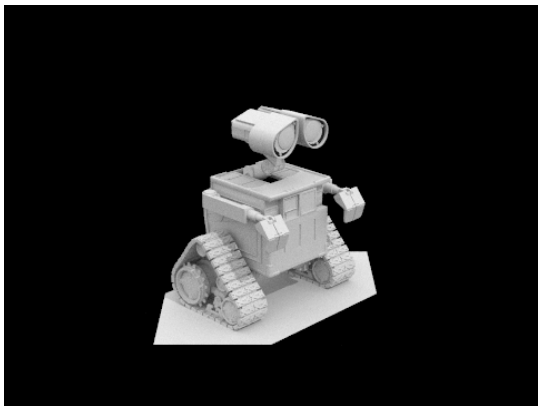
Part 4: Global Illumination (20 Points)

Walk through your implementation of the indirect lighting function.

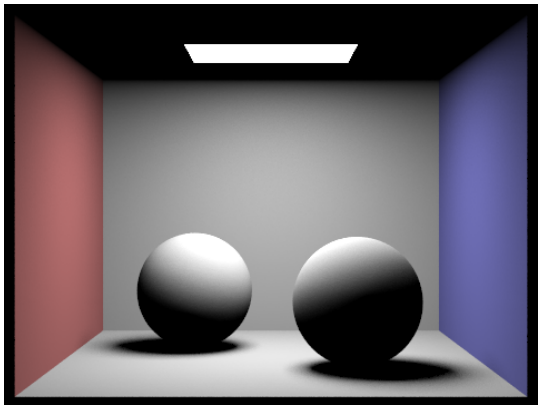
To implement full global illumination, I used my direct lighting function and added support for it to render with indirect lighting. By doing this, we can keep track of the light that is reflected on surfaces in the whole scene. I first updated my est_radiance_global_illumination() function by calling zero_bounce_radiance() first and then adding the result of at_least_one_bounce_radiance() to it. To implement at_least_one_bounce_radiance what I did was check if the depth is greater than 1. If it is, we want to calculate the radiance contribution from one_bounce_radiance() function and set it to L_out. Then I used coin_flip() to generate a random number. If the random number is less than .5 and the depth of the ray is not the same as the max ray depth, we want to terminate by returning L_out. This randomness termination is Russian Roulette. We also want to terminate if the depth of the ray is less than 1. If none of the conditional statements didn't terminate the function, we want to sample a Bidirectional Scattering Distribution Function (BSDF) value using the surface intersection point. Afterwards, I want to create a new ray in the direction of the sampled direction. Afterwards, I check if the new created ray intersects the scene and if it does, we recursively calculate the radiance contribution from the bounced light with the at_least_one_bounce_radiance() function, adding the result to L_out. I finished it off by normalizing the radiance contribution with the PDF of the sampled direction and returning L_out.

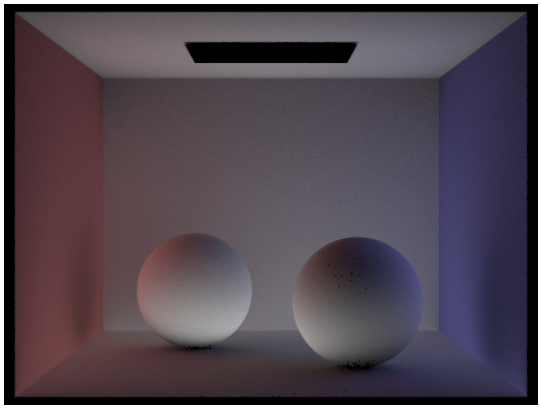
Show some images rendered with global (direct and indirect) illumination. Use 1024 samples per pixel.


CBbunny.dae

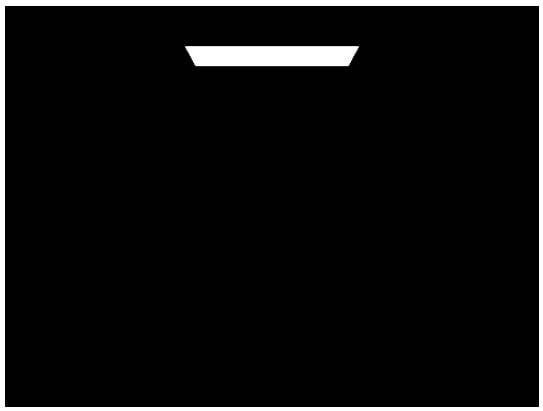

wall-e.dae

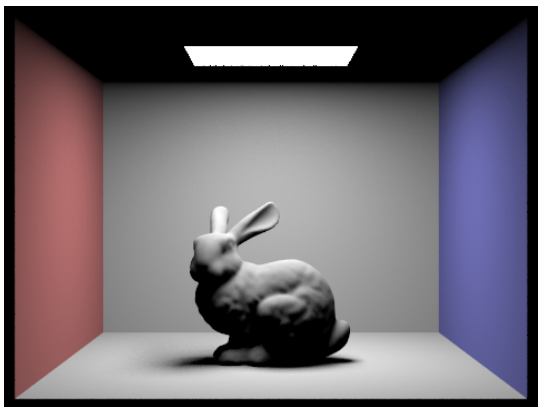
Pick one scene and compare rendered views first with only direct illumination, then only indirect illumination. Use 1024 samples per pixel. (You will have to edit PathTracer::at_least_one_bounce_radiance(...) in your code to generate these views.)

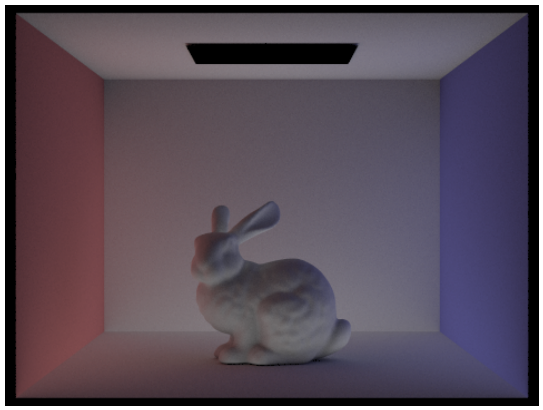

Only direct illumination - CBspheres_lambertian.dae

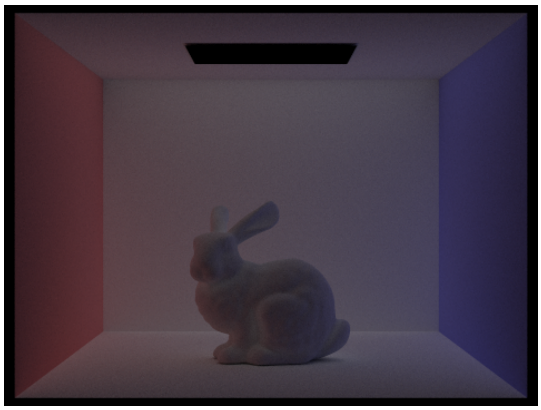

Only indirect illumination - CBspheres_lambertian.dae

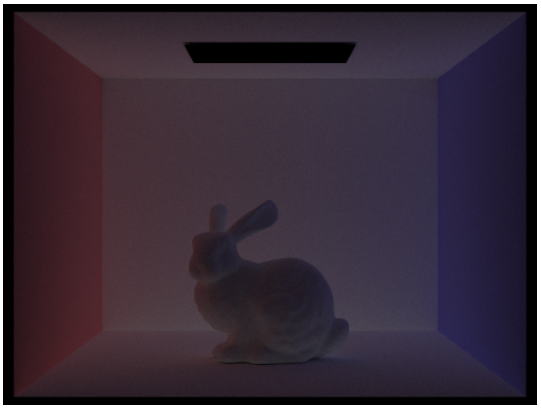
For CBbunny.dae, render the mth bounce of light with max_ray_depth set to 0, 1, 2, 3, 4, and 5 (the -m flag), and isAccumBounces=false. Explain in your writeup what you see for the 2nd and 3rd bounce of light, and how it contributes to the quality of the rendered image compared to rasterization. Use 1024 samples per pixel.

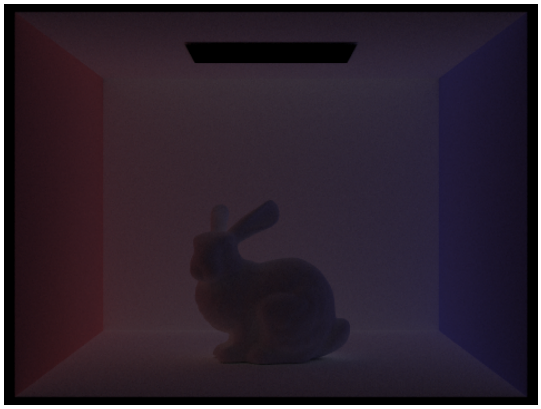

max_ray_depth = 0 (CBbunny.dae)


max_ray_depth = 1 (CBbunny.dae)

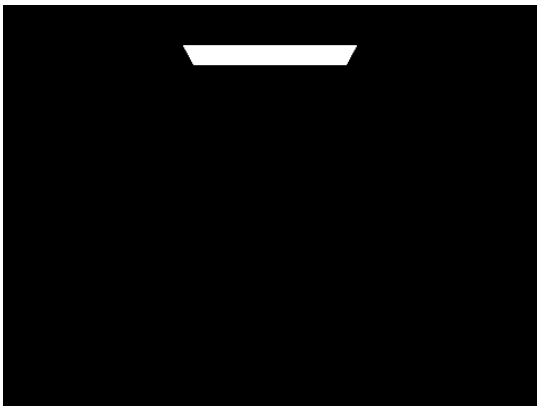

max_ray_depth = 2 (CBbunny.dae)

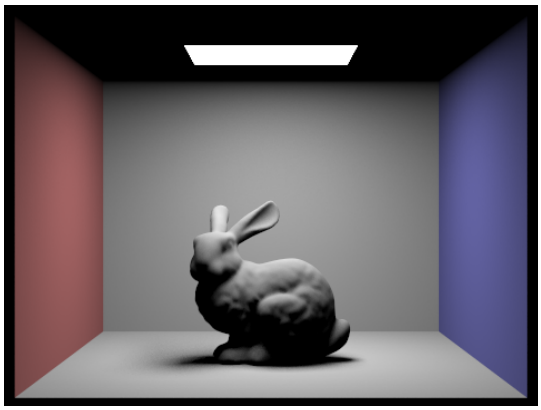

max_ray_depth = 3 (CBbunny.dae)

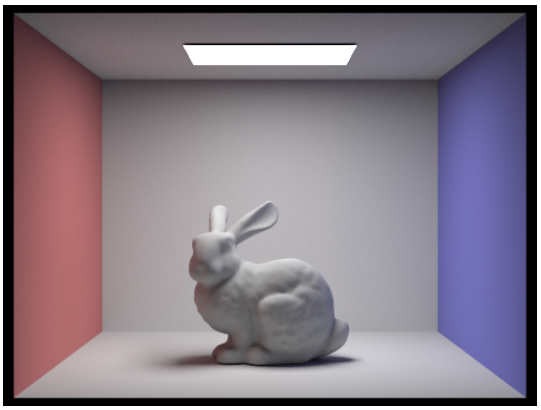

max_ray_depth = 4 (CBbunny.dae)

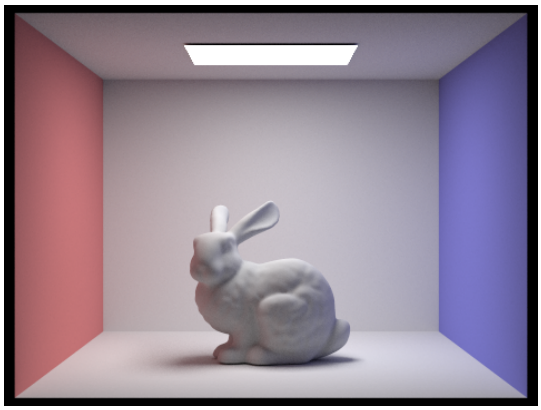

max_ray_depth = 5 (CBbunny.dae)

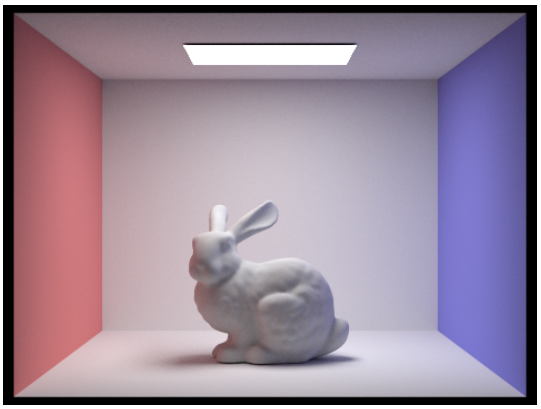
For CBbunny.dae, compare rendered views with max_ray_depth set to 0, 1, 2, 3, 4, and 5(the -m flag). Use 1024 samples per pixel.

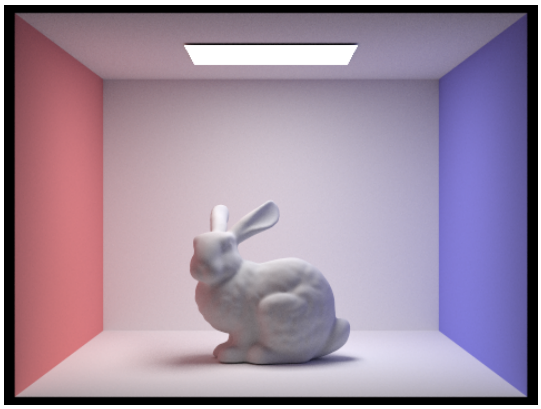

max_ray_depth = 0 (CBbunny.dae)


max_ray_depth = 1 (CBbunny.dae)


max_ray_depth = 2 (CBbunny.dae)

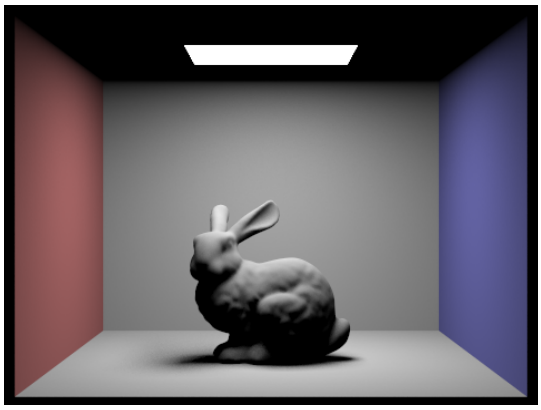

max_ray_depth = 3 (CBbunny.dae)

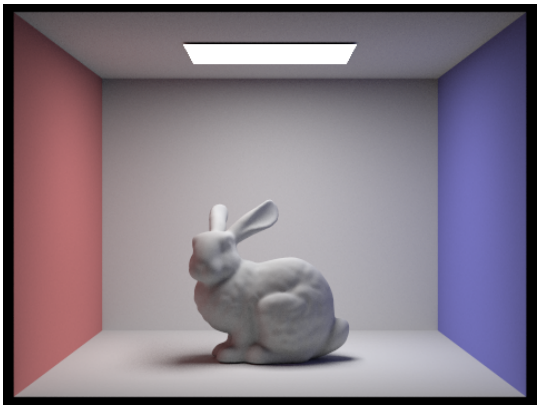

max_ray_depth = 4 (CBbunny.dae)

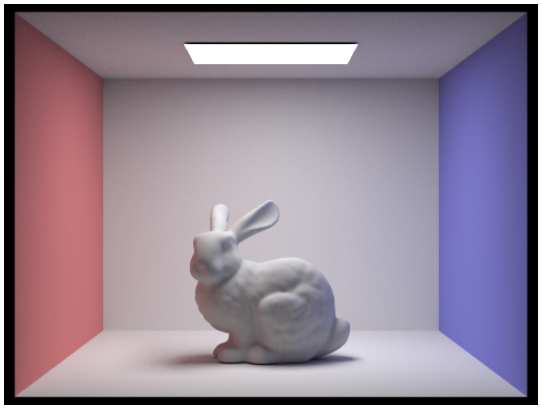

max_ray_depth = 5 (CBbunny.dae)

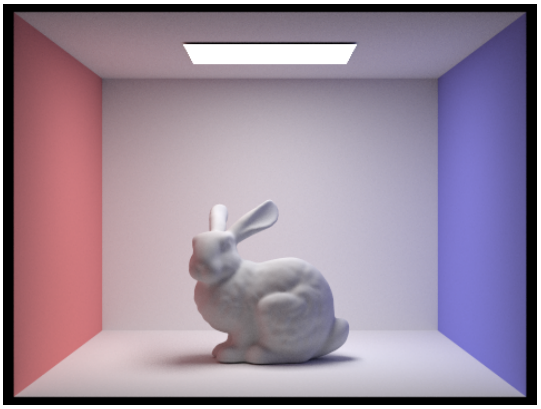
For CBbunny.dae, output the Russian Roulette rendering with max_ray_depth set to 0, 1, 2, 3, 4, and 100(the -m flag). Use 1024 samples per pixel.

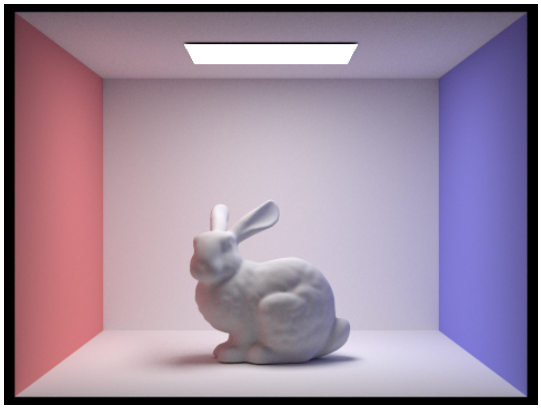

max_ray_depth = 0 with Russian Roulette(CBbunny.dae)


max_ray_depth = 1 with Russian Roulette(CBbunny.dae)

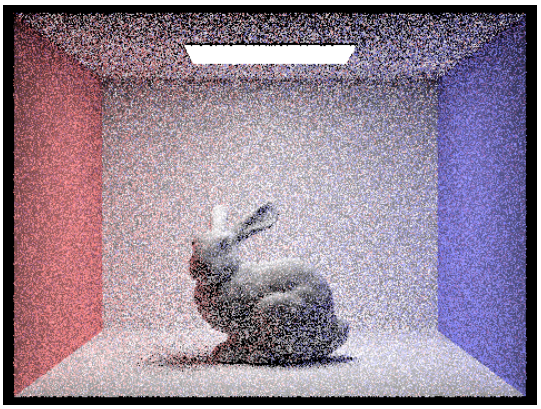

max_ray_depth = 2 with Russian Roulette(CBbunny.dae)

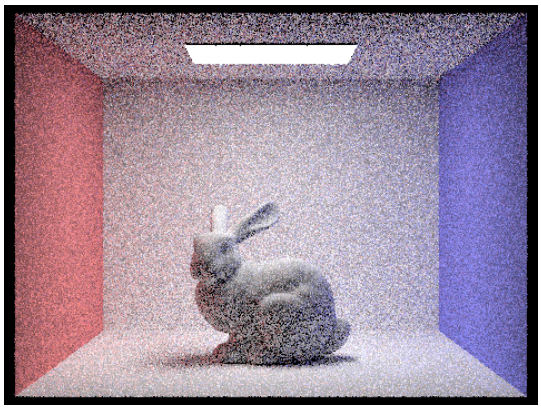

max_ray_depth = 3 with Russian Roulette(CBbunny.dae)

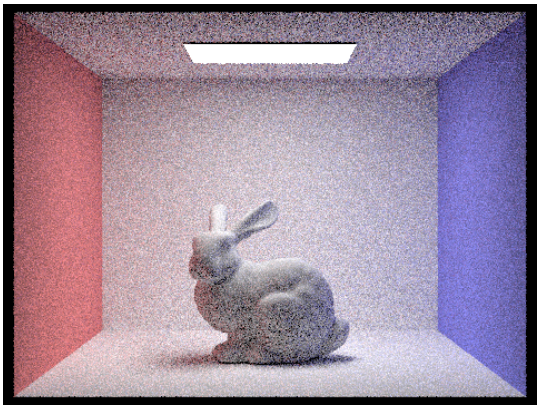

max_ray_depth = 4 with Russian Roulette(CBbunny.dae)

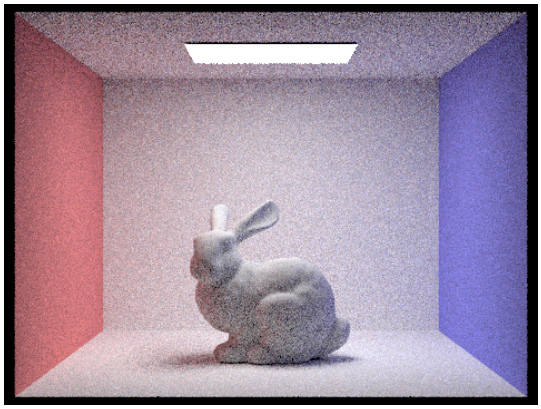

max_ray_depth = 5 with Russian Roulette(CBbunny.dae)

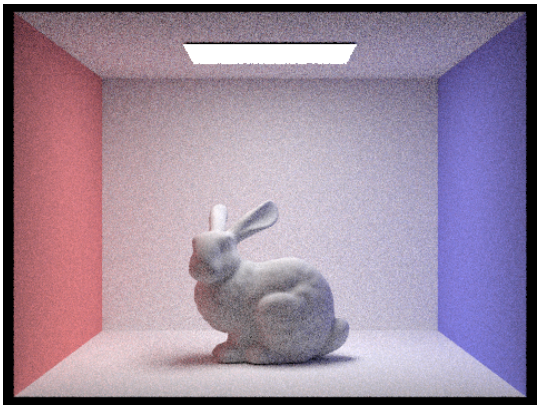
Pick one scene and compare rendered views with various sample-per-pixel rates, including at least 1, 2, 4, 8, 16, 64, and 1024. Use 4 light rays.

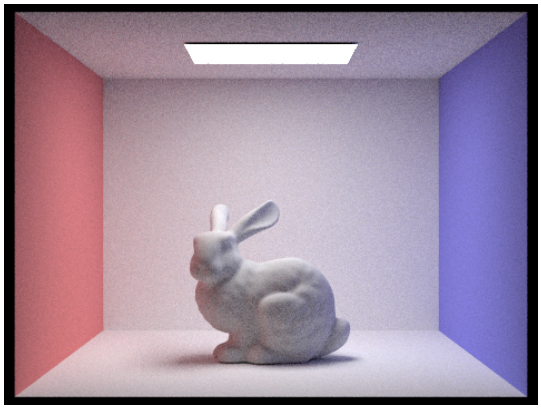

1 sample per pixel (CBbunny.dae)

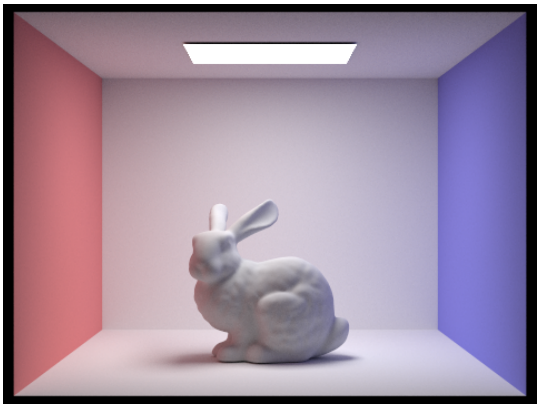

1 sample per pixel (CBbunny.dae)


2 sample per pixel (CBbunny.dae)


4 sample per pixel (CBbunny.dae)


8 sample per pixel (CBbunny.dae)


16 sample per pixel (CBbunny.dae)

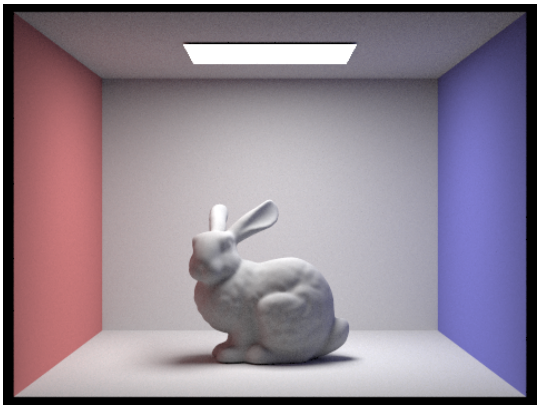

1024 sample per pixel (CBbunny.dae)

Part 5: Adaptive Sampling (20 Points)

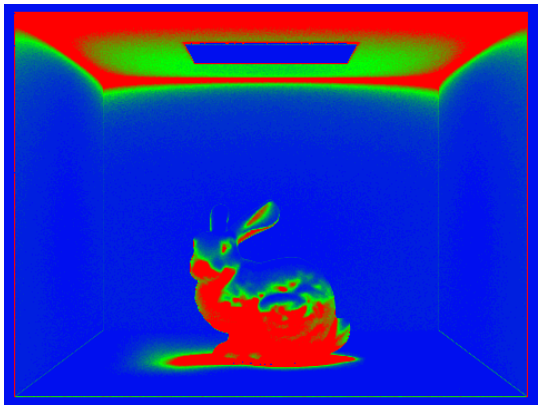
Explain adaptive sampling. Walk through your implementation of the adaptive sampling.

To implement adaptive sampling, I update the raytrace_pixel() function. The first thing I did was create two double variables, s1 and s2. Next, when I iterate through all the samples, I will get the illuminance of the sample and set it to s1. I also want to square the illuminance and set it to s2. With s1 and s2, I can then calculate the mean and the standard deviation if we reach the sample threshold (samples % samplesPerBatch == 0 and samples > 0). I then want to divide the standard deviation by the total number of samples currently in that iteration. The result is the pixel convergence and if it is less than or equal to (MaxTolerance * mean), the loop terminates early because the pixel converges. If it doesn't it continues the loop.

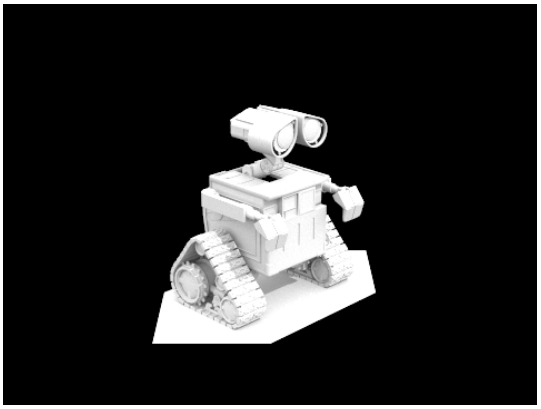
Pick two scenes and render them with at least 2048 samples per pixel. Show a good sampling rate image with clearly visible differences in sampling rate over various regions and pixels. Include both your sample rate image, which shows your how your adaptive sampling changes depending on which part of the image you are rendering, and your noise-free rendered result. Use 1 sample per light and at least 5 for max ray depth.



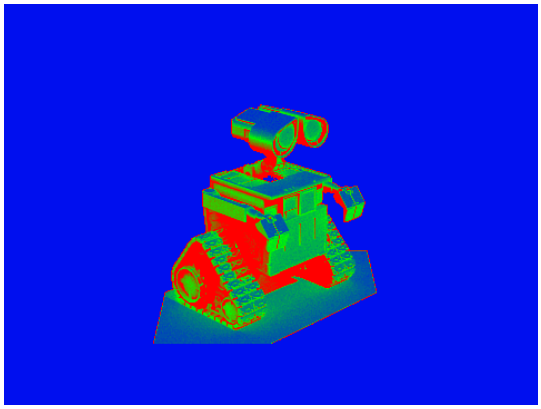
Rendered image (CBbunny.dae)



Sample rate image (CBbunny.dae)



Rendered image (wall-e.dae)



Sample rate image (wall-e.dae)