Homework 3 Writeup Webpage Link

Homework 3 Writeup

Project Overview:

Overall, this project was quite difficult, and required going through many previous discussions, lectures slides, and office hours for debugging help. However, it was quite rewarding seeing how all of the different ray generation, BVH/bounding box, and illumination techniques influenced lighting and coloring of an image in different ways.

Part 1: Ray generation and scene intersection

Part 1 was relatively straightforward, as we were just following the equations given in the first 2 tasks. We wrote out all of the matrix muliplicationa dn linear algebra on paper in order to reverse engineer the transform matrix for camera space to world space using the given coordinate points (more details in the part 1 writeup). We played around with the constants in order to debug, until our output image matched the staf solution. In our opinions, ray-triangle intersection was the most difficult to implement.

Part 2: BVH Stuff

In this part we were tasked with creating the BVH and intersecting it. The BVH construction algorithm is clearly explained in Part 2's more in-depth writeup. I will go more in-depth into the BVH intersection. In order to check the intersection of a ray with the BVH, we used the built-in has_intersection function of the primitive class. For the intersect, if the ray misses the node's bounding box, we return. Otherwise, we know it intersects, and we check if the node is a leaf. If so, we intersect the primitive with the ray. Otherwise, we recurse on the node's left and right children until we reach a leaf node. Overall, these functions were relatively simple to implement. However, we ran into segfaulting initially, which we believe was due to accessing a member's field of the primitive that we weren't supposed to. Now, we were able to render images in fractions of time compared to previous.

Part 3: Direct Illumination

This task directly helped us render images with better shadows, and greatly reduce noise of pixels in the image. The direct lighting function implemnentation walkthrough is listed in the writeup for that part in more depth. We will walk thrgouh some issues we faced with Task 3, uniform hemisphere sampling. We initially ran into issues where the direct lighting with uniform hemisphere sampling returns an entirely black room, no matter what. the first thing in our debugging process we did was check the Monte carlo estimator function, and each of its componenets. We realized that the Li value from get_emission() was constantly returning 3D vector of only 0 values, which would explaint the lack of illuminance. We initially believed that this error could be from a previous step, like in the ray intersection or BVH construction. This is because one of the groups we spoke to didn't have a bounding box on the light on the ceiling. Unfortunately, making some changes with a TA didn't change this. In the end, we were not correctly doing something in task 2.1 with the intersect checking of a ray, leading to the program believing that none of the rays were intersecting with the bounding box, so the image would render nothing with light. After this was fixed, we were able to move onto importance sampling, which was relatively simple and similar to the previous implementation of hemisphere sampling, just changing a few parameters with the new rays we were creating.

Part 4: Global Illumination

This task helped us better spread out light equally around an image, rather than have it all focused in a few main locations. The main implemntation was done in at_least_one_bounce and russian roulette. In these tasks, we finished est_radiance_global_illumnation in order to estimate the global illumination arriving at a point from a specific direction, considering multiple bounces of light, which we achieved by recursively calling at_least_one_bounce, which traced rays in sampled directions based on the BSDF at the hit point and computed radiance contributions. The logic of termination and further recursion is mentioned in more depth below. In the task after, we implemented Russian ROulette in order to randomly terminate rays. This technique helped us unbiasedly improve effieincy of the program, while maintaining

accuracy in estimating radiance. The russian roulette algorithm involved a coin flip function in order to randomly decide whether to continue recursing forward, or terminate.

Part 5: Adaptive Sampling

Part 1:

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

In order to generate the ray, the pipeline starts with the image coordinates (x, y) in Image Space. We created a 3x3 transform based on inverse mapping a set of linear equations from points in the spec (creating matrices for mapping center (0.5,0.5)->(0,0), bottom corner (0,0)->(-tan(0.5*hFov, -tan(0.5*vFov))), etc.). Then, we multipled the ray's x, y to camera space by matrix multiplying with the transform matrix. Finally, we multiply by c2w to change the direction vector to worldspace, and then normalize it. Finally, we create a ray based on the camera pos and calculated direction vector, set the min_t and max_t to nClip and fClip, and then return that ray. From there, we update the pixel in a SampleBuffer with the integral of radiance over the pixel. To make this integral estimation, we loop num_samples number of items, generate a sample, create a newX and newY, which are the translated coordinates of the sample. Then, we sum a vector with all of these values passed into est_radiance_global_illumnation, and return the average vector after updating the samplebuffer pixel. image coords -> ray in world space (generate_ray) -> update pixel in SampleBuffer w/ integral of radiance over pixel

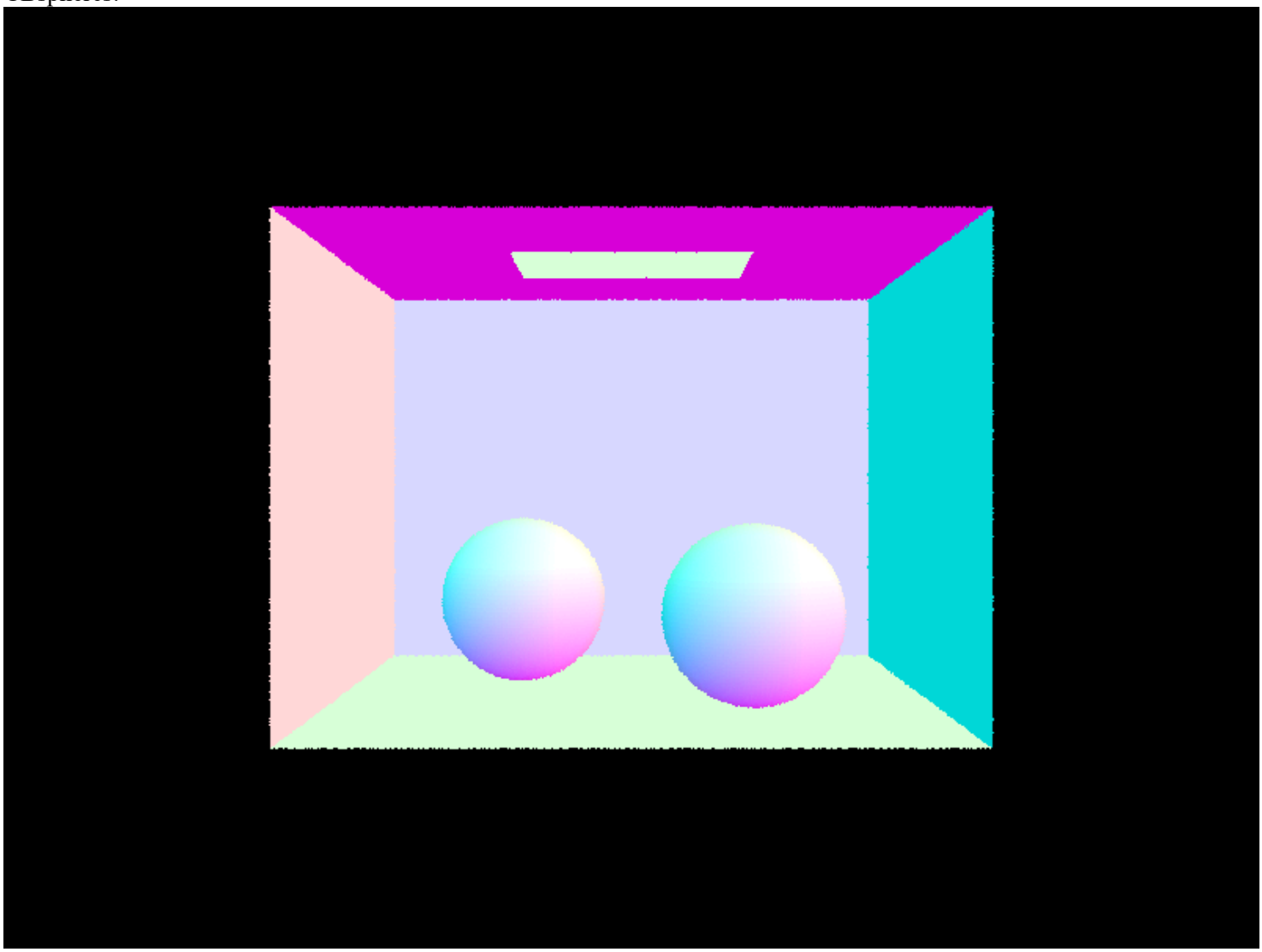
Sphere intersection: The main functionality about ray-sphere intersection was based on the lecture slides 9-21. We are testing if a ray intersects a Sphere by creating float values a, b, c, and temp, where temp = the quadratic formula equation. The first hypothetical ray sphere intersection is at -b + sqrt(...), and the second hypothetical intersection is at -b - sqrt(...); then we assign the inputted double pointers t1 and t2 to the corresponding min(first, second) and max(first, second) respectively, and return true. If the value b^2 - 4ac < 0, then we know that there is no intersection between the ray and sphere, so we return false. If there is an intersection, then we set the Intersection *i data: n (normal) with equation ray's origin + t*ray's direction - center of sphere location (the normalized vector pointing from the sphere center to the intersection point) primitive, bsdf, and t.

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

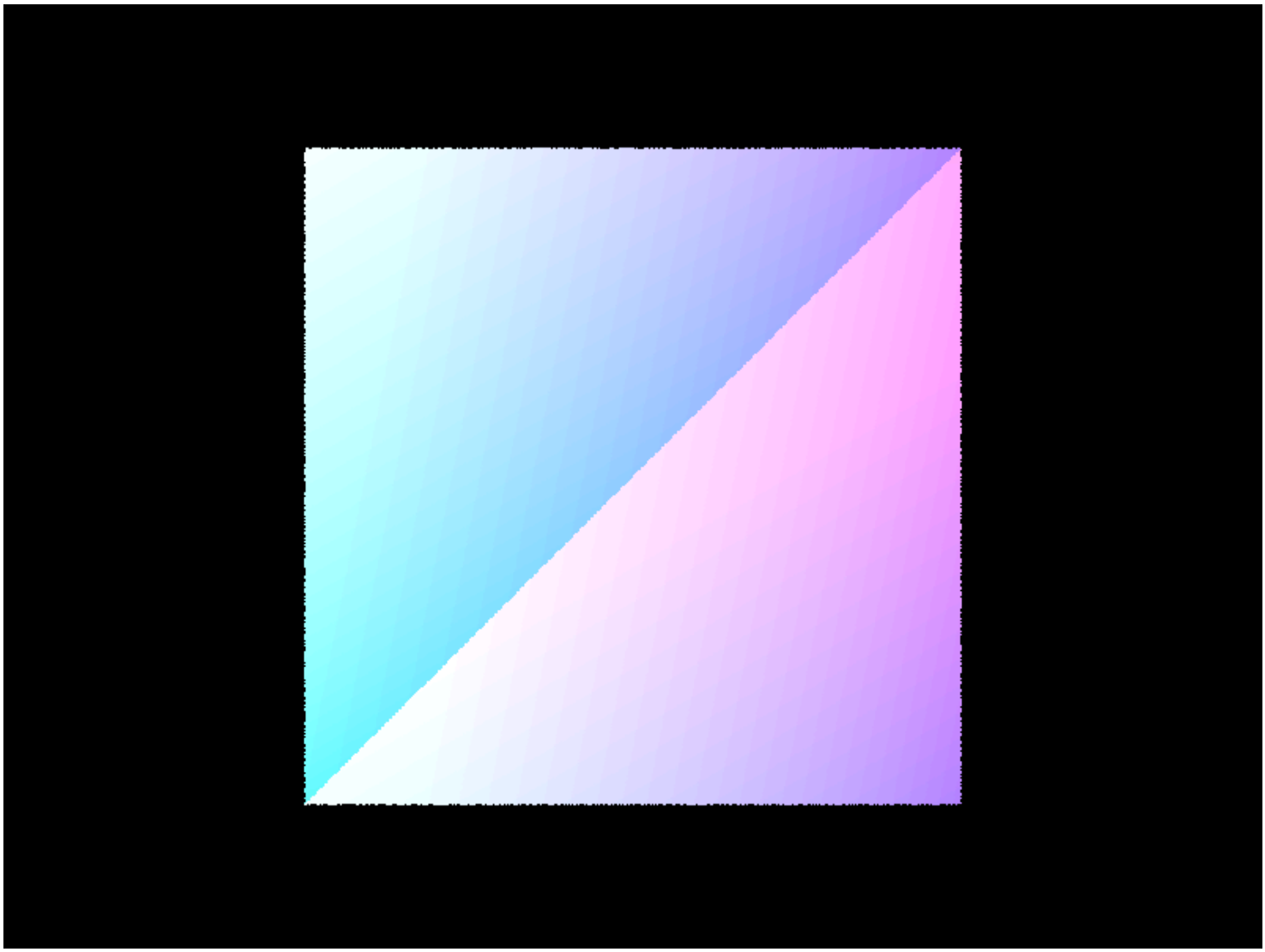
In order to do ray-triangle intersection, we used the Moller Trumbore algorithm to calculate E1, E2, S, S1, and S2, essentially the barycentric coordinates of the ray's intersection with the triangle. Based on these values, we output the vector of params [t, b1, and b2]. Finally, we compare that ray's parameterization, t, ensuring many conditions, such as being >= 0, being between min_t and max_t. If these conditions hold, we know that the ray intersects with the triangle. If there is an intersection, we now set the parameters of the ray and the intersection. For the ray, we update its max_t parameter to equal t (the nearest interesction), so that we can limit the space of future interesctions that we check. For the interesction, we update its t value, its normal as a weighted sum of the 3 vertice's normals, the primitive, and the bsdf. Finally, we are able to render the CBempty.dae room, so that the walls and ceiling are existent.

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

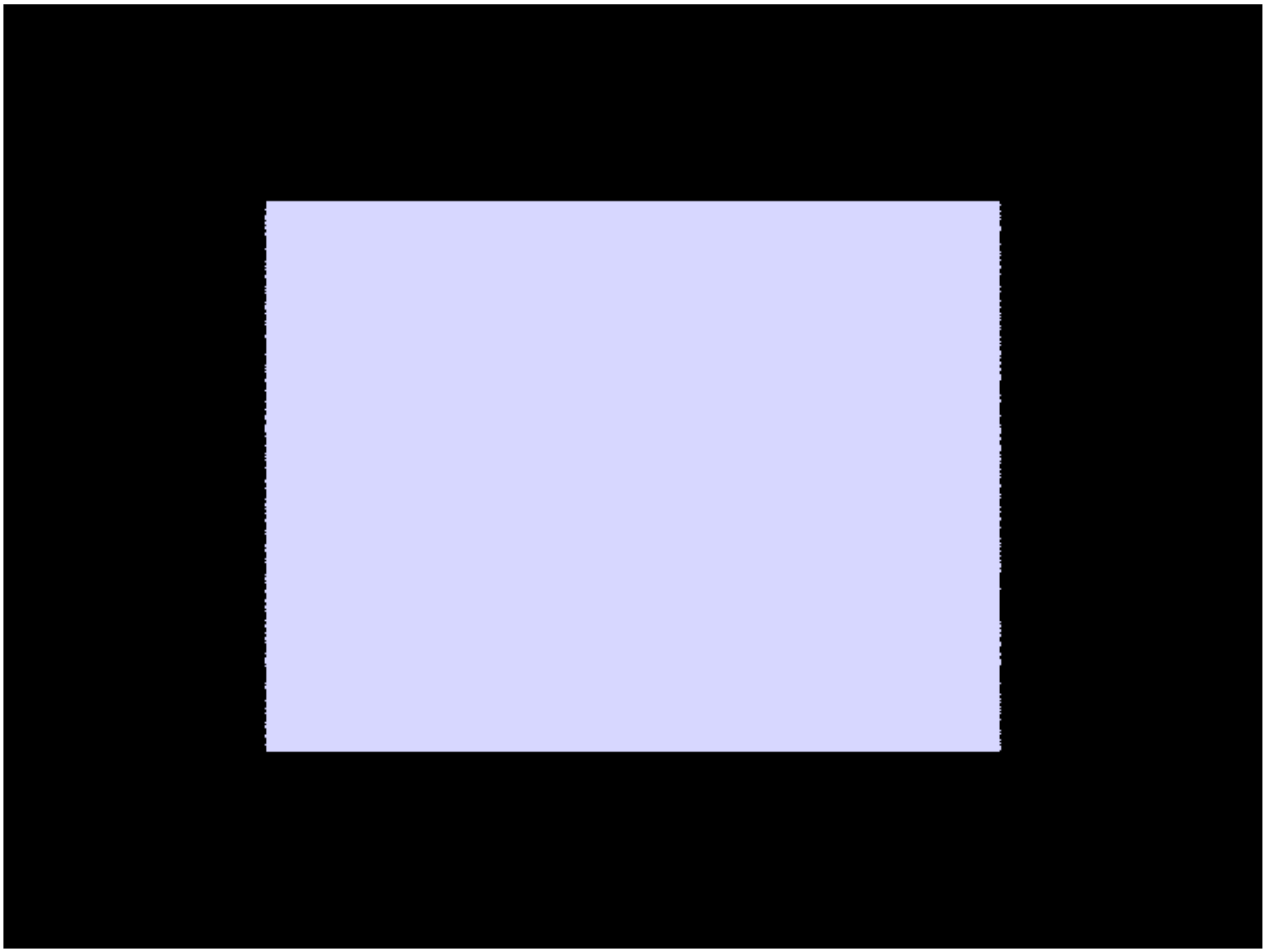
CBspheres:



Cube:



plane:



Part 2:

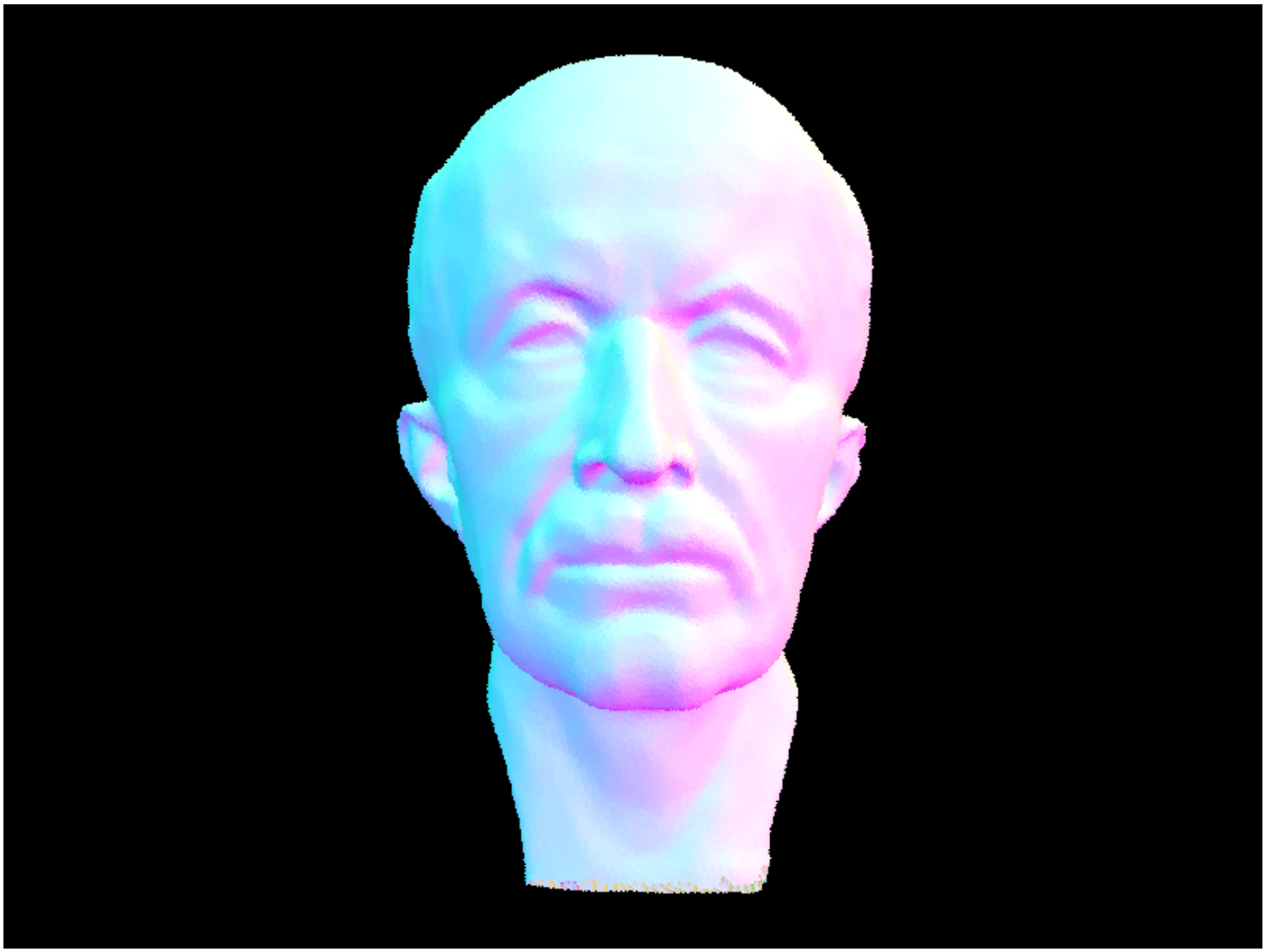
Walk through your BVH construction algorithm. Explain the heuristic you chose for picking the splitting point.

Our BVH construction algorithm started by creating a single BVH node (the root node). THen, we used recursion to build up the tree. The recursive case was if the count of vector of Primitives was less than the max_leaf_size, then we would set the start and the end of a BVHnode to that vector's start and end, and return the node. In the recursive case, we first find the best axis to split on by a simple for loop that increments 3 counters (aboveX, aboveY, and aboveZ) for a Primitive if its x, y, and z centroid coordinates are above the bounding box's centroid coordinates. Then, we take each counter, and calculate a temp value: aboveX - (total_count - aboveX) for each axis, in order to see which axis is the most "balanced" (most balanced for us is the temp value closest to 0). Then, we create 2 new vector of Primitive pointers left and right. For the selected axis, we iterate over the entire inputted vector of Primitives, and assign each Primitive pointer to left if its centroid is less than the bounding box's center for that axis we chose, and right if its centroied is larger than the bounding box's center for the axis we chose. The heuristic we chose for picking the spliting point was the axis with the most "even split", as we wanted the left and right split to give us the most information gain.
One thing we noticed was that we were initially segfaulting, and that was because we didn't handle the case where one of the children nodes remains empty (left is empty || right is empty). To alleviate this issue of infinite recursion, we checked if either vector is empty, and moved a Primitive from the "full" side to the empty.

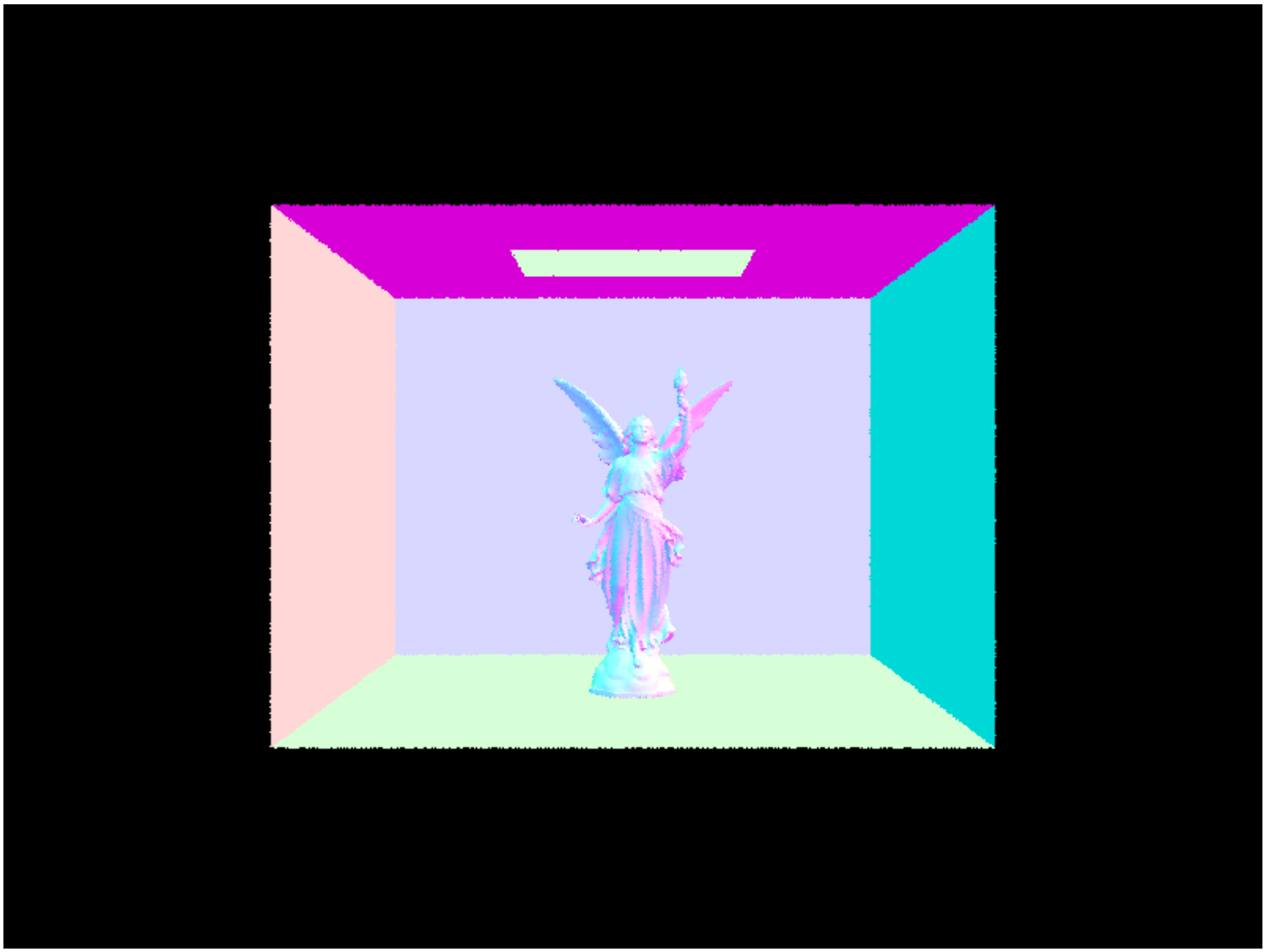
Show images with normal shading for a few large .dae files that you can only render with BVH acceleration.

Cow:

0.03 seconds
maxplanck:



0.3 seconds
lucy:



0.1 seconds

Compare rendering times on a few scenes with moderately complex geometries with and without BVH acceleration.

With BVH acceleration, it took **0.03 seconds for cow**, **0.3 seconds for maxplanck**, and **0.1 seconds for lucy**. Without BVH acceleration, it took **5 seconds for cow**, **47 seconds for maxplanck**, and **232 seconds for lucy**. Clearly, there was a huge difference in the time it took to render with and without BVH acceleration; using BVH acceleration, the cow image rendered ~166 times faster, the maxplanck rendered ~16.6 times faster, and the lucy image rendered ~2320 times faster. Based on our results, this shows that the BVH acceleration GREATLY helps speedup the rendering of images with many triangles-- for example, the lucy image has hundreds of thousands of triangles, and the speedup gain from BVH acceleration is greatly bigger than the maxplanck image, which only has tens of thousands of triangles.

Part 3:

Walk through both implementations of the direct lighting function.

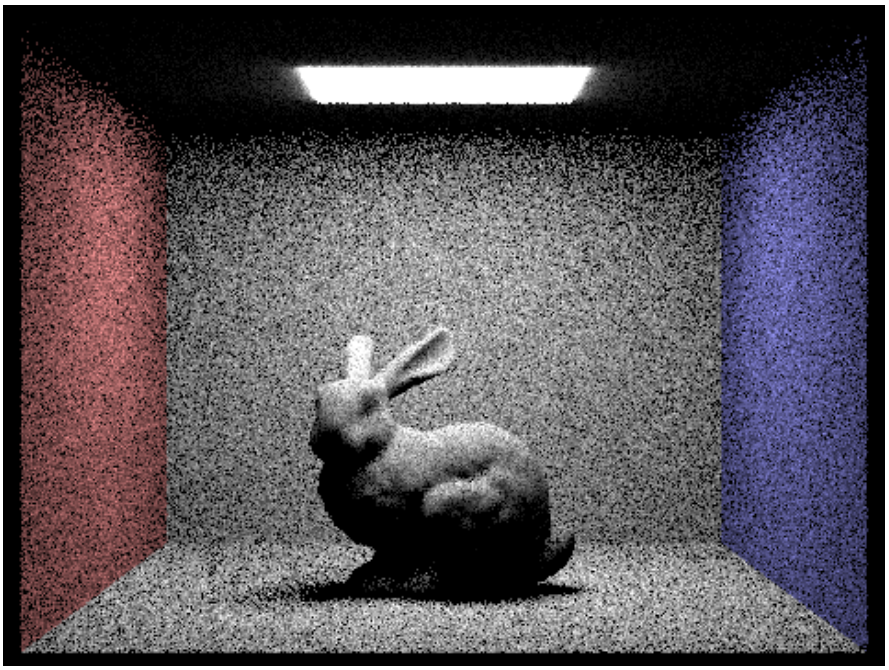
Direct Lighting with Uniform Hemisphere Sampling The first direct lighting function we we implemented was done through uniformly sampling in a hemisphere, following the Monte Carlo estimator. To implement this, we sampled num_samples number of times: first getting the emission value of the surface material, w_i, then getting the output of diffusing lambertian BSDF passing in w_i and w_out. Then, we create a new ray with direction d = o2w*w_i. If this ray's direction intersects with the bvh of the light source, then we contribute its new Li value (new intersection's emission value), and fill in the other two constant values cos and pdf of the estimator. Finally, we add this to our existing sum. If the ray

doesn't intersect with the bvh of the light source, we continue. Finally, we return the runing sum divided by the number of samples (average).

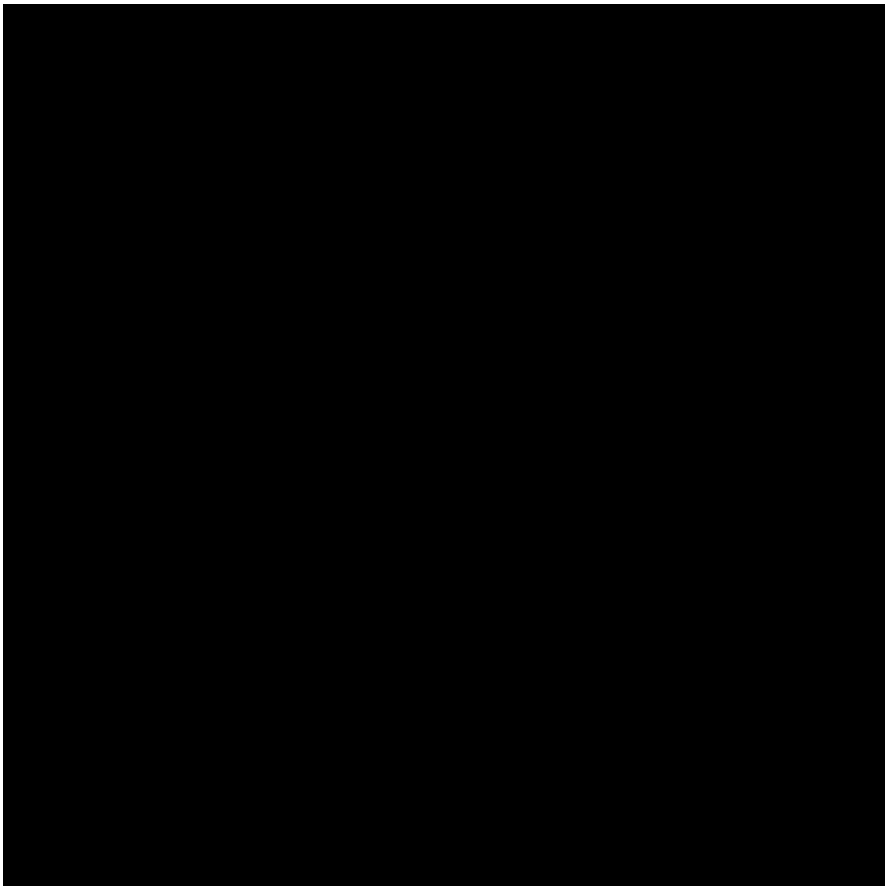
Direct Lighting by Importance Sampling Lights The second direct lighting function we implemented was importance sampling, meant to sample all points directly, with the advantage that we can render images with only point lights. For each light source in a scene, we similarily sample num_samples times: Similarily to hemisphere sampling, we create a new ray, and calculate the Li emission sample = light->sample_L and the output of f(w_out, w_i) for the Monte Carlo estimator. The difference is that now, the ray's max_t parameter = the distance from the surface to the light - EPS_F and the ray's direction = w_i (we set these ray parameters). After creating and setting the new ray's parameters, the rest of the implementation is essentially the same as hemisphere sampling: check if the ray intersects with the light source BVH, and add to our running sum. And finally, divide by the number of samples to get the average. Now, the images rendered have less noise, and the scenes are brighter from point lights.

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

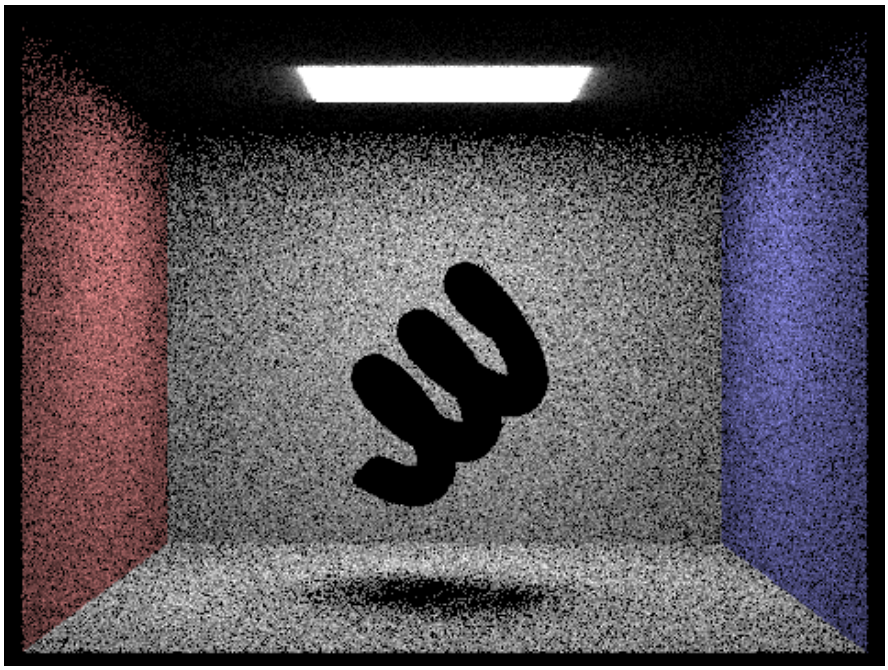
Uniform Hemisphere Sampling:



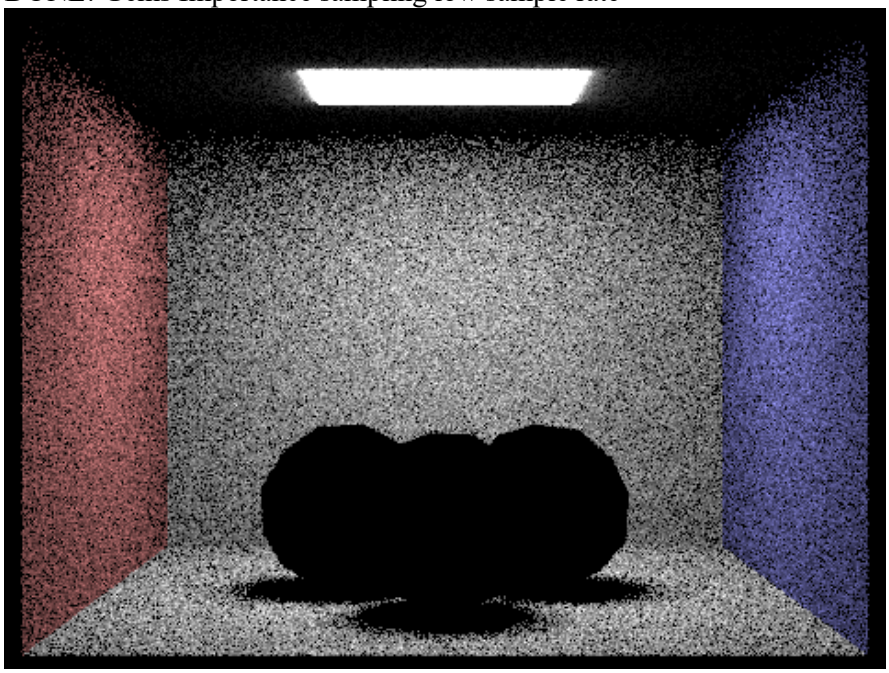
CBunny Hemisphere sampling Low sample rate



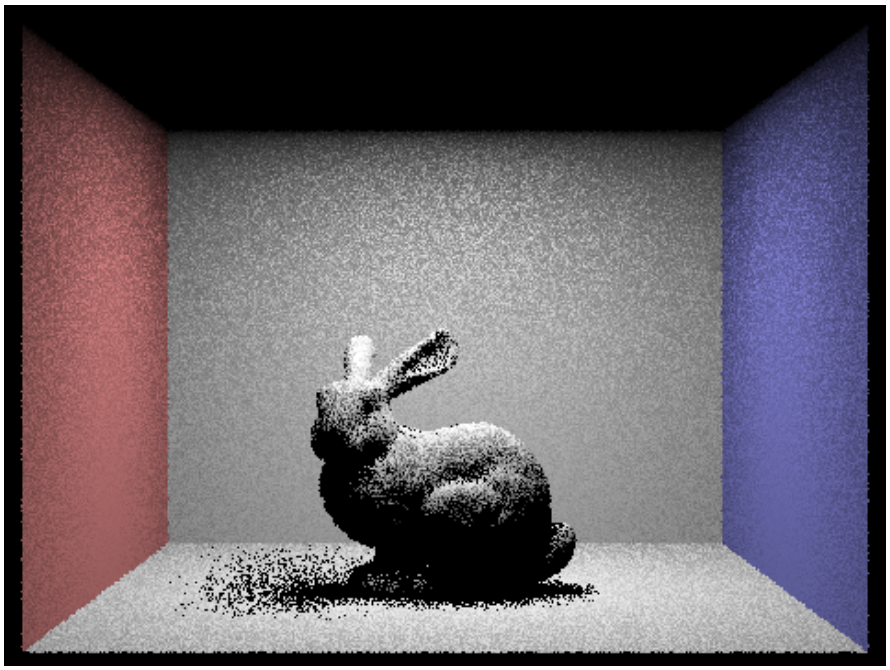
Dragon Hemisphere sampling high sample rate



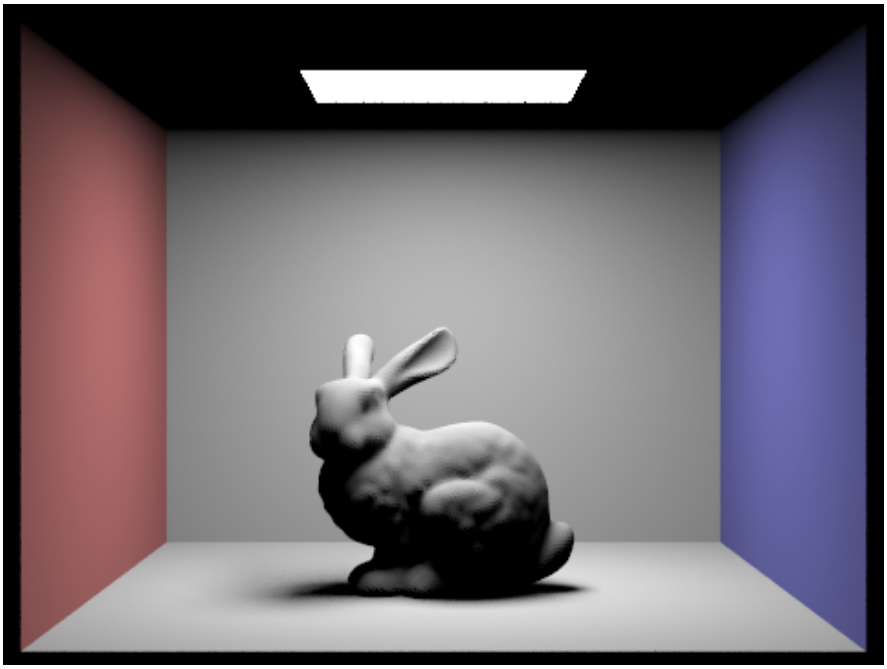
Coil Hemisphere sampling low sample rate
DONE! Gems Importance sampling low sample rate



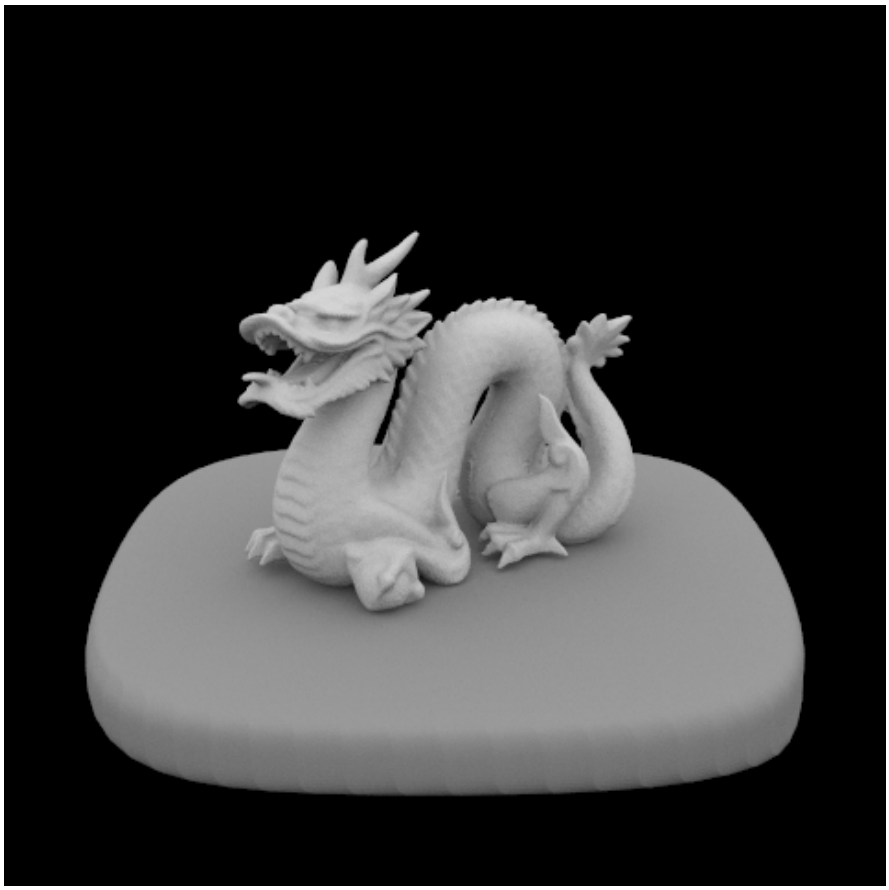
Importance sampling



CBunny Importance sampling Low sample rate



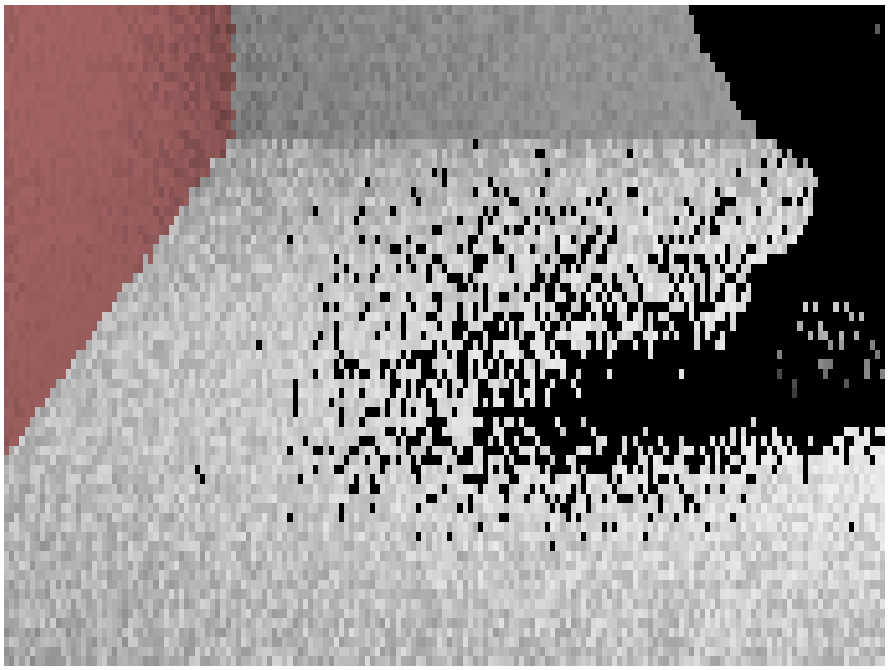
CBunny Importance sampling High sample rate



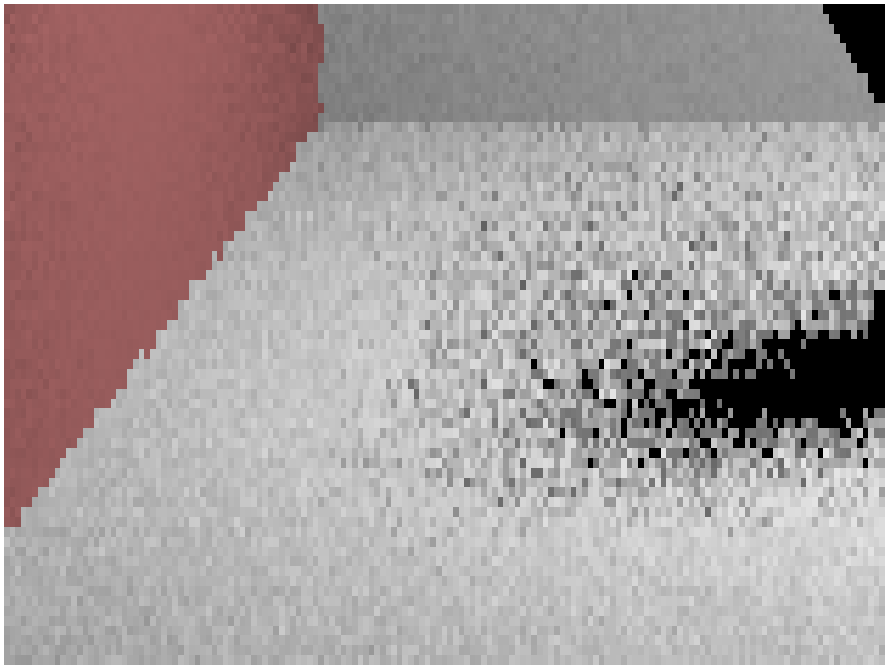
Dragon Importance sampling high sample rate

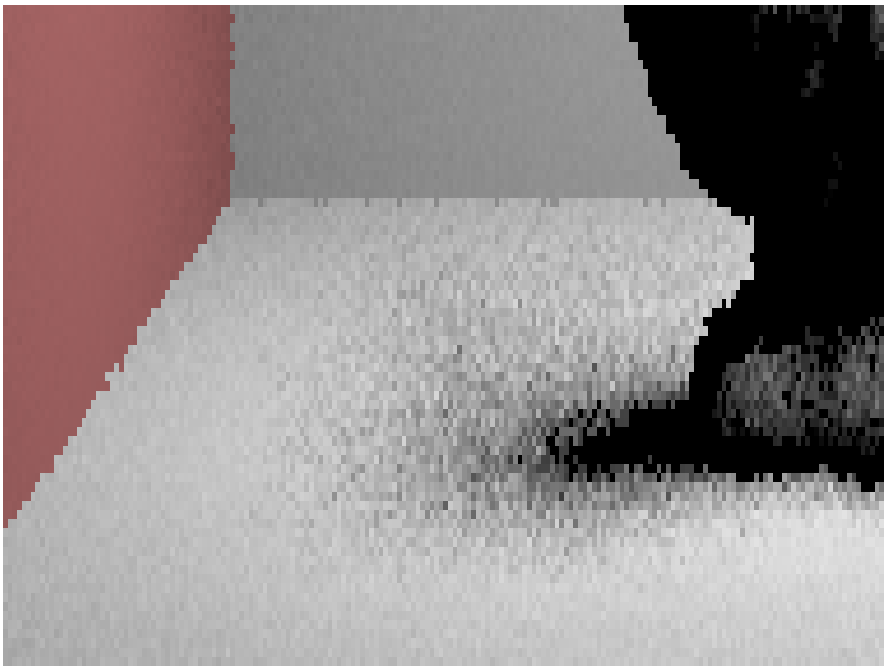
Focus on one particular scene with at least one area light and compare the noise levels in soft shadows when rendering with 1, 4, 16, and 64 light rays (the -l flag) and with 1 sample per pixel (the -s flag) using light sampling, not uniform hemisphere sampling.

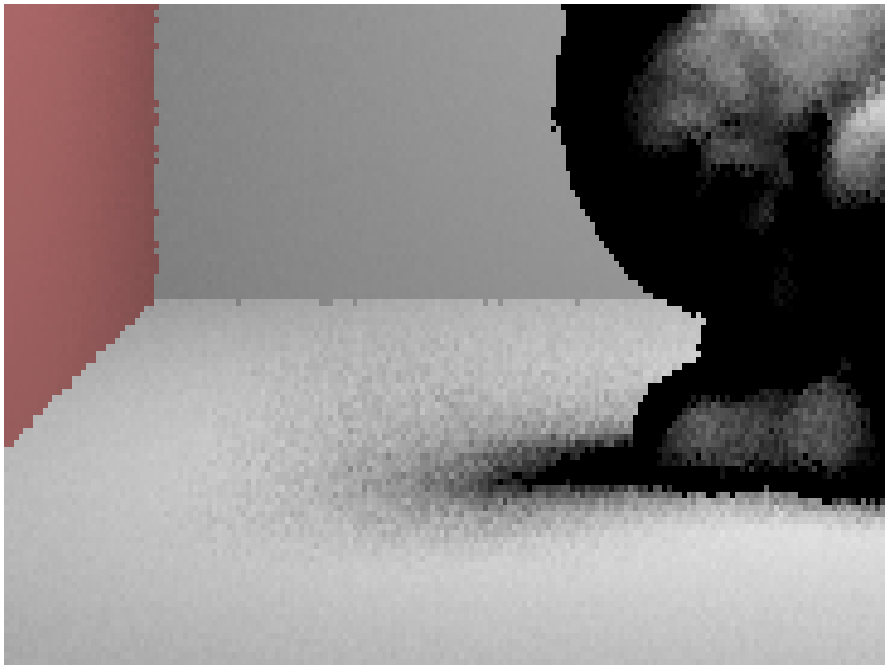
1 light ray:



4 light rays:



16 light ray: 

64 light rays: 

There is much less noise in soft shadows as we increase the amount of light rays used. In low light ray renderings, like with 1 or 4 light rays, there are fewer samples taken to determine the lighting. Thus, there is obvious gradients of black shadow of light. As we increase the light rays used, like 16 or 64, we can see smoother transitions of indirect lighting effects, leading to accurate shadows.

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

There is typically more noise in uniform hemisphere sampling than lighting sampling. While both images are noisy, this difference can be apparently seen in CBunny Importance sampling Low sample rate vs. CBunny Hemisphere sampling low sample rate. The hemisphere sampling bunny has way more black pixels spotted around the background of the image. On the other hand, lighting sampling prioritizes biasing significant light sources, which is interesting because the light source is completely blacked out.

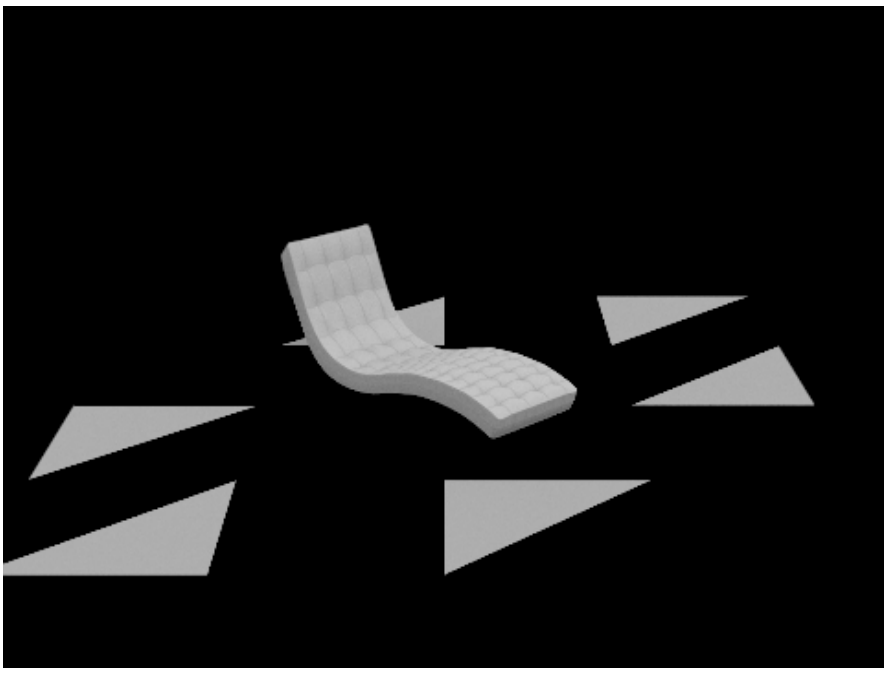
Part 4:

Walk through your implementation of the indirect lighting function.

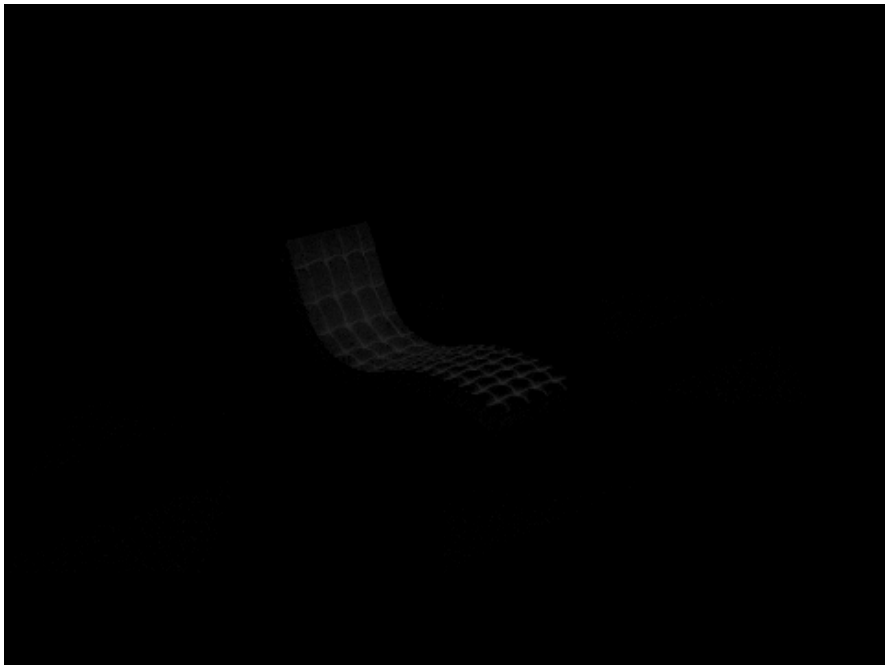
For the indirect lighting function, the overall functionality is similar to direct light sampling, we just also recurse to reach more bounce cases. We are using the ray depth (r.depth) as the recursive condition, and the base case is when ray depth is 0, and we decrement ray depth in each recursive call. There's also a separate condition where if isAccumBounces is true/false, and if it's false, we check if ray depth <= maxRayDepth. In the recursive case, we generate a new ray based on a point on the mesh, and sample outgoing rays, getting the lighting from rays that intersect, and dimming them based on distance. We sum each lighting output from each recursive bounce, and return the sum in the end.

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

Bench direct illumination (1 bounce):

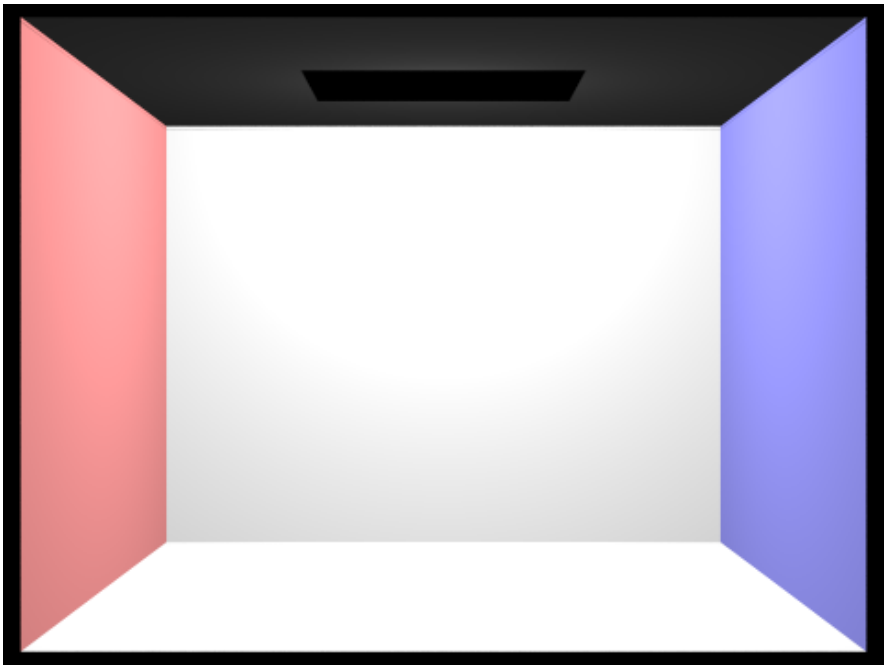


Bench indirect illumination (3 bounce):

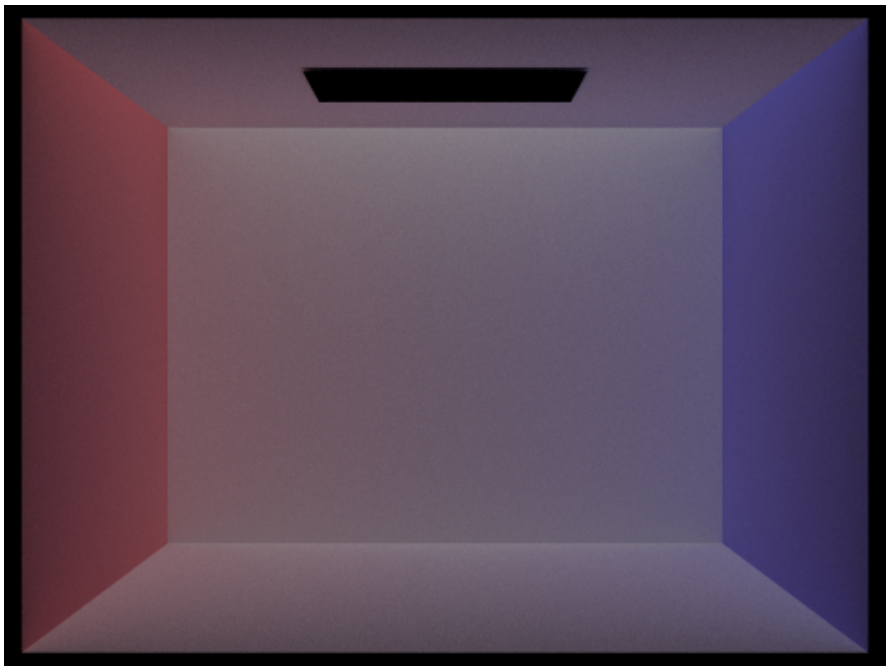


Bench indirect illumination (5 bounce):

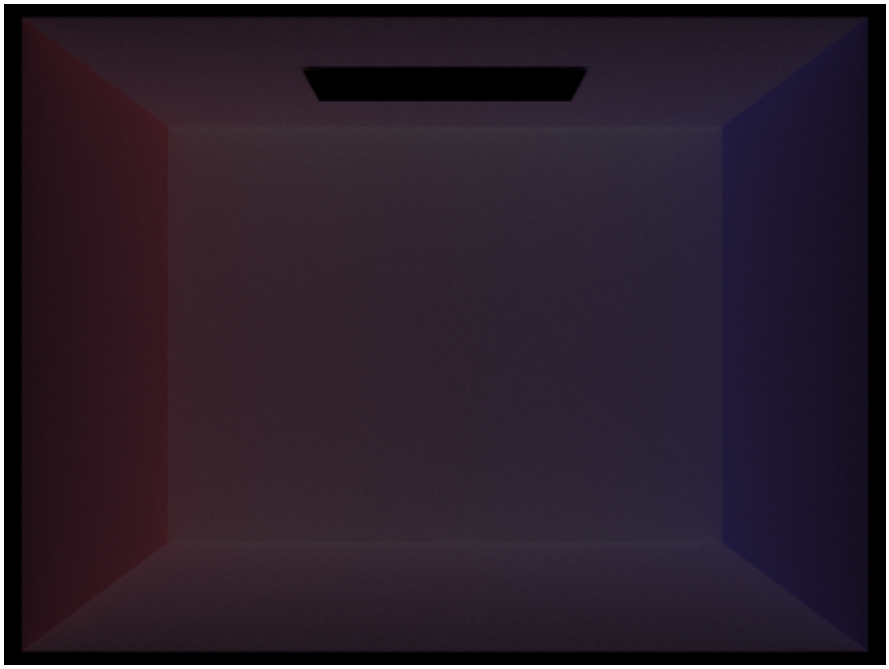
EMpty direct illumination (1 bounce):

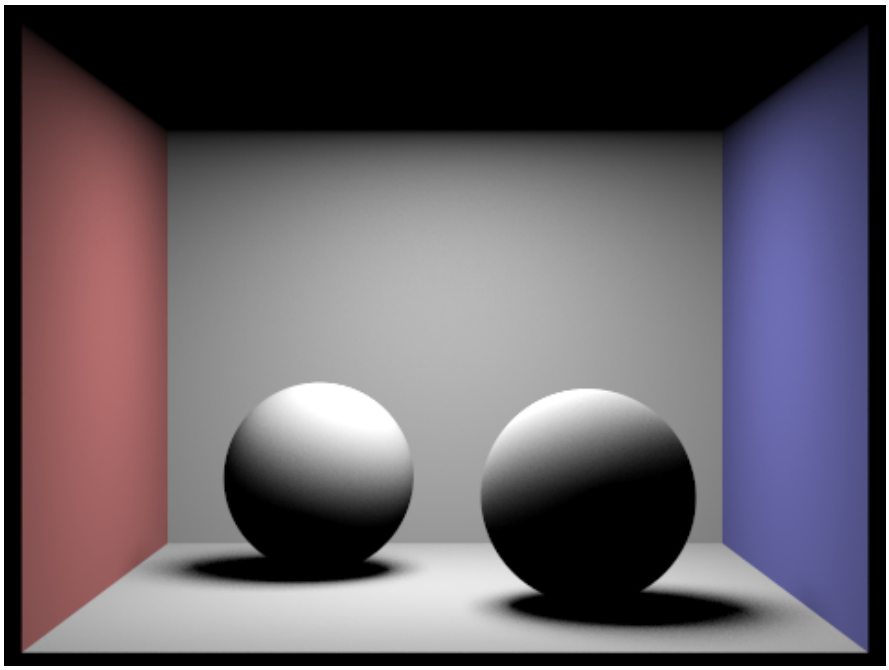


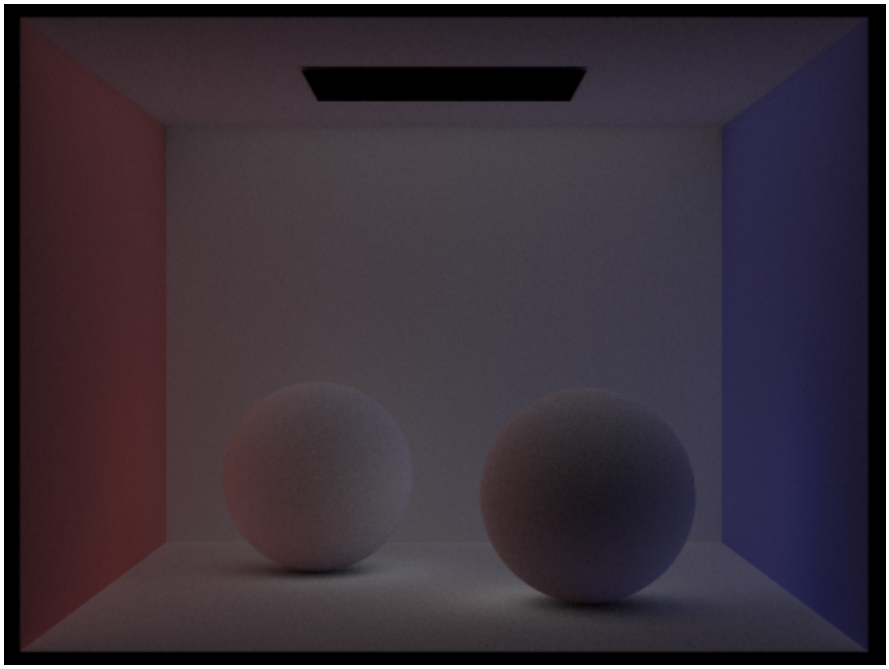
Empty indirect illumination (3 bounce):

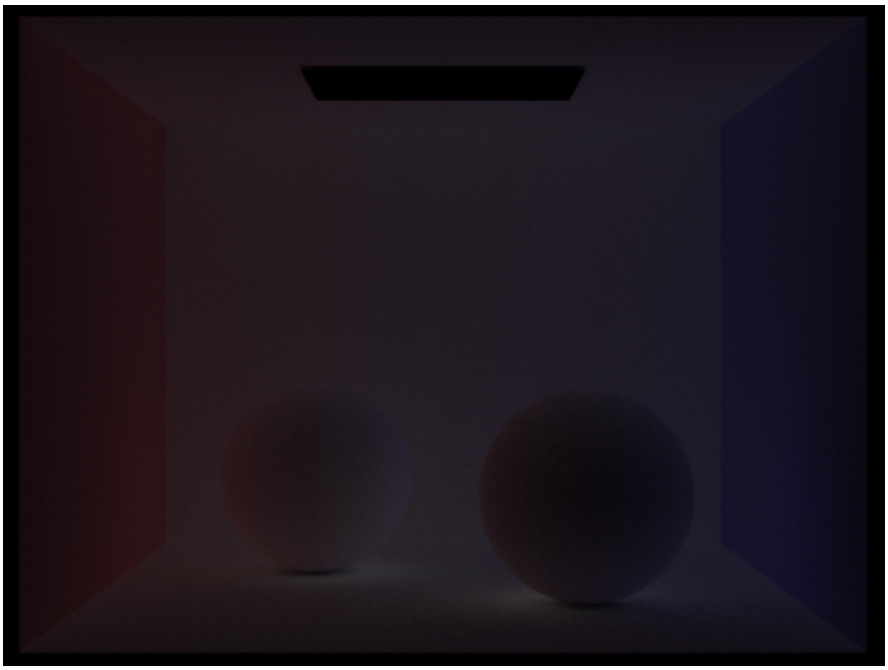


Empty indirect illumination (5 bounce):



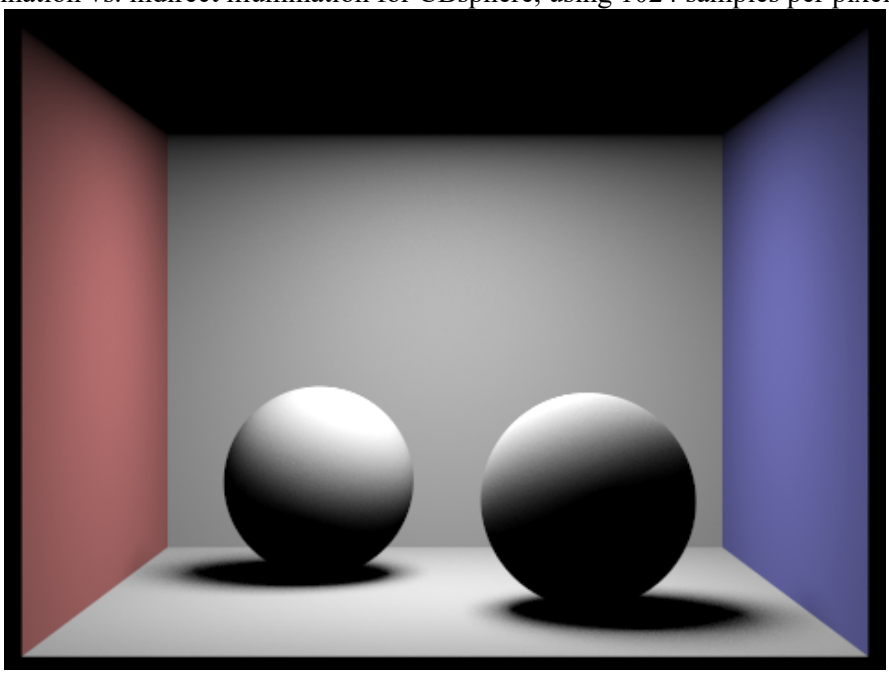
CBSphere direct illumination (1 bounce): 

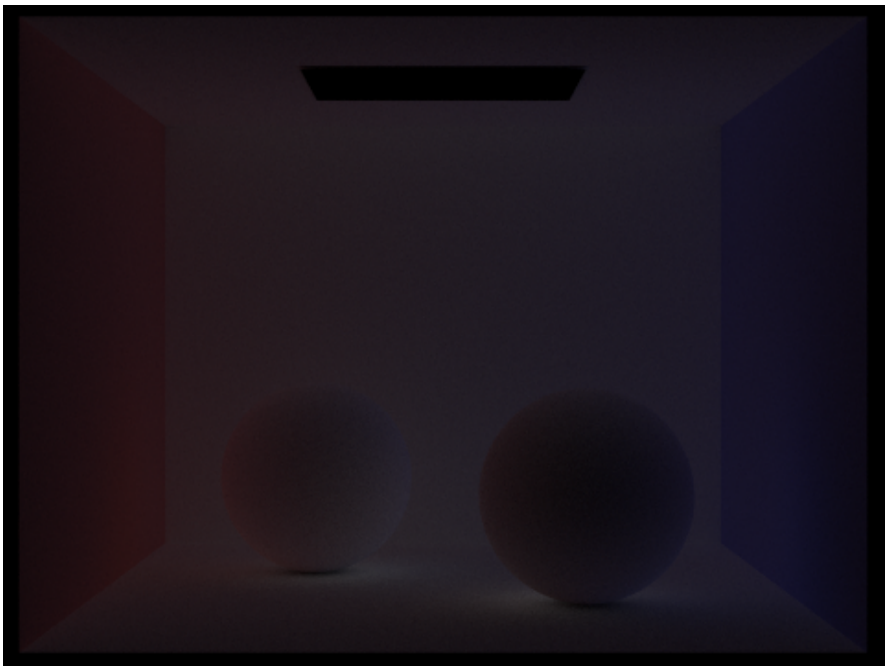
CBSpheres indirect illumination (3 bounce): 

CBSpheres indirect illumination (5 bounce): 

Pick one scene and compare rendered views first with only direct illumination, then only indirect illumination. Use 1024 samples per pixel.

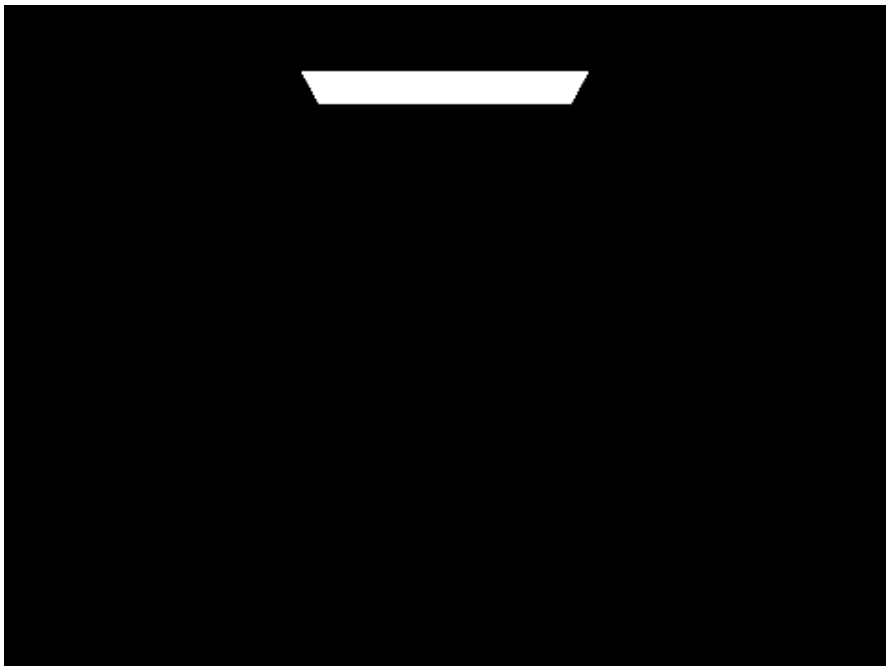
We will compare the scene of direct illumination vs. indirect illumination for CBsphere, using 1024 samples per pixel.

CBSphere direct illumination (1 bounce): 

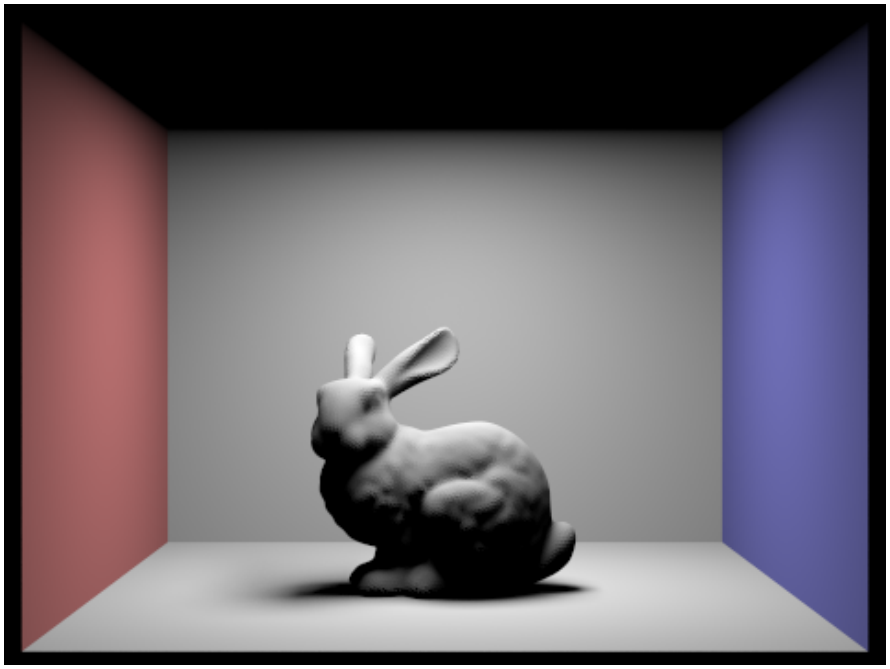
CBSpheres indirect illumination (5 bounce): 

Two of the main differences between direct illumination and indirect illumination is that direct illumination has much sharper lighting contrasts and patterns, while indirect illumination has smoother lighting gradients. We can see in the direct illumination, there's almost 3 light bands on the left sphere (white, grey, black), indicating the sharpness of lighting contrast, whereas the indirect illumination (3 bounces) doesn't have apparent differences next to one another. Second, indirect illumination adds better light ambiance with the environment, which can be seen where the walls and floor background around the spheres also aren't as sharply contrasted with the spheres because of the bounces.

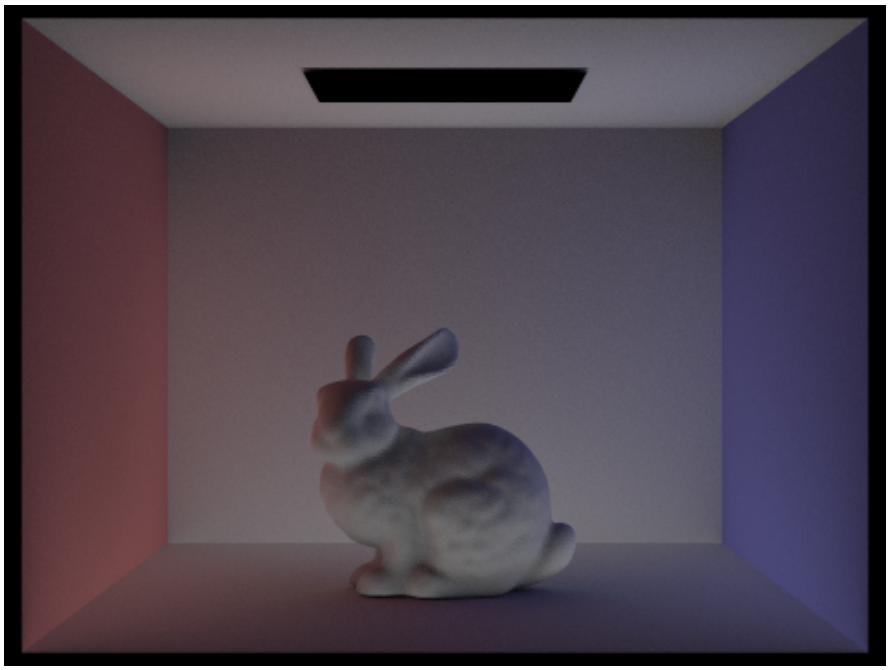
For CBbunny.dae, render the mth bounce of light with max_ray_depth set to 0, 1, 2, 3, 4, and 5 (the -m flag), and isAccumBounces=false. Explain in your writeup what you see for the 2nd and 3rd bounce of light, and how it contributes to the quality of the rendered image compared to rasterization. Use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



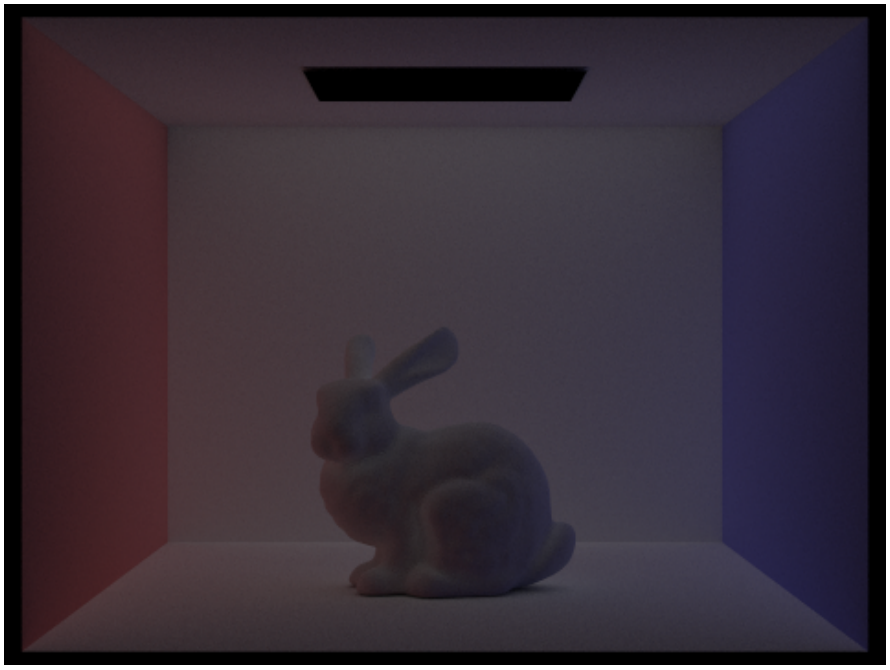
0th bounce of light



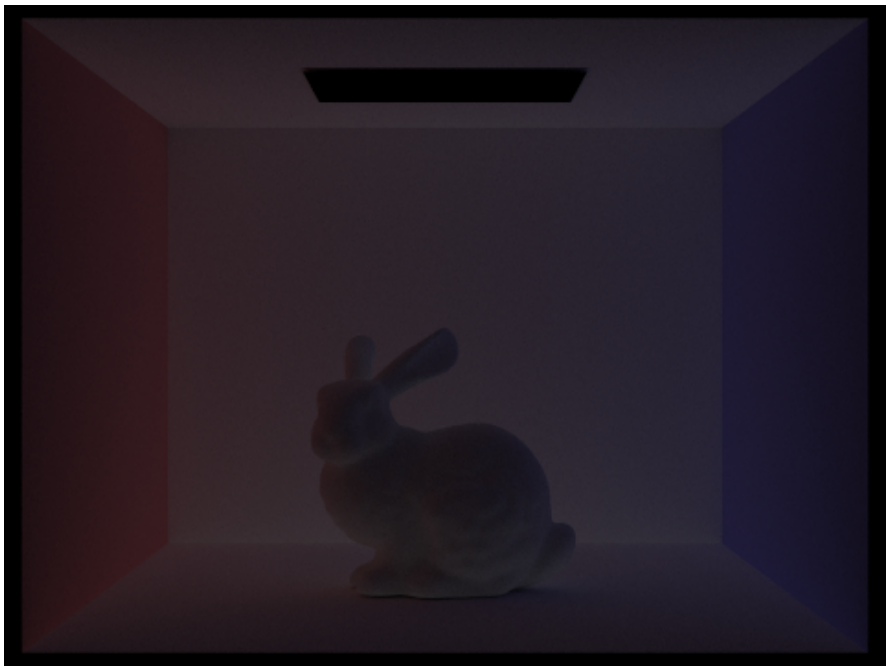
1th bounce of light



2th bounce of light



3th bounce of light

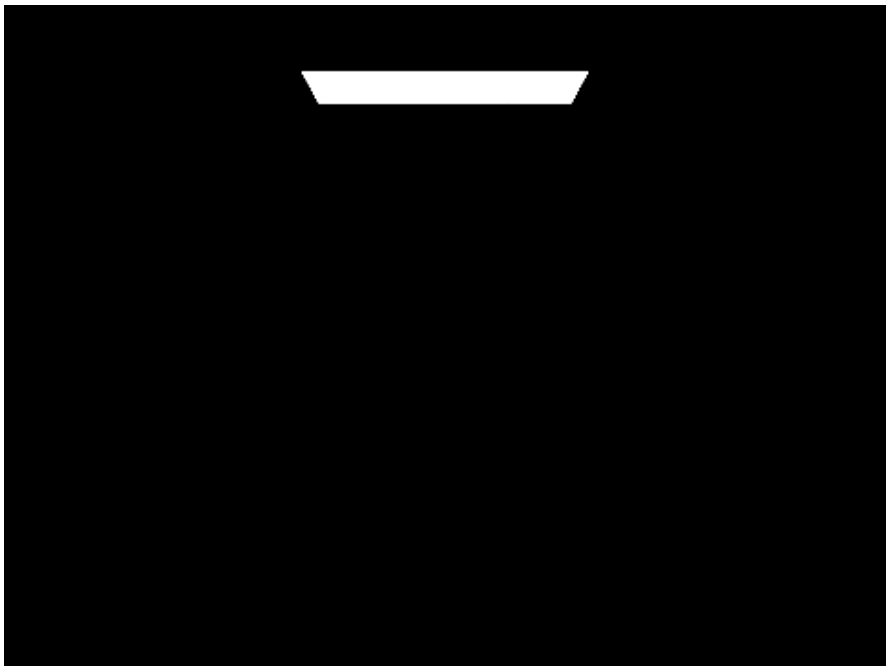


4th bounce of light

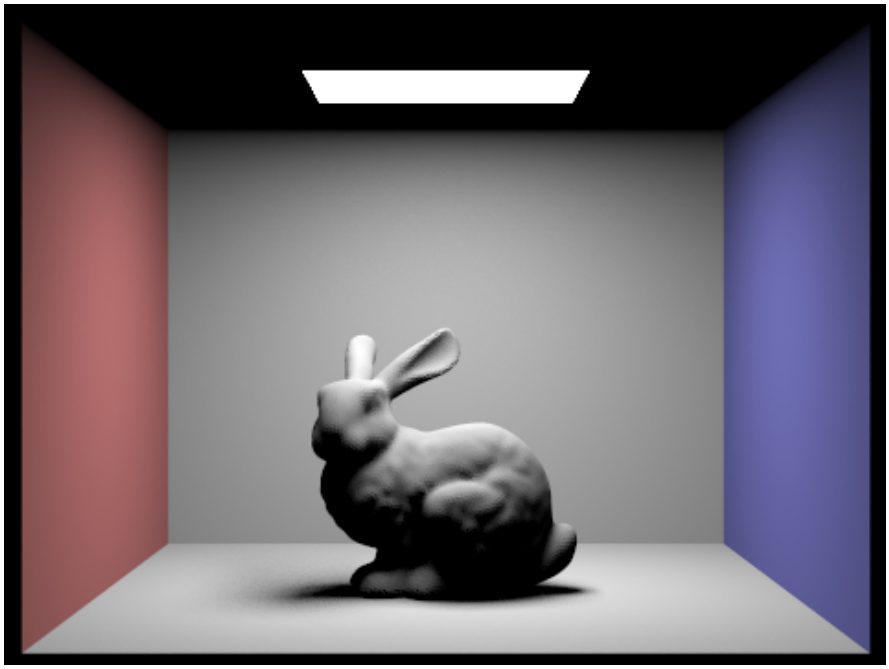
5th bounce of light

For the 2nd and 3rd bounces of light, it looks like there is light coming from underneath the bunny, and that the bunny's body is glowing, ilke adding color bleeding to the scene. This is because the bunny is reflecting some light off of the wall, and the wall is reflecting some light/color off of the bunny, which adds better color complexity, and decreases noise as compared to rasterization.

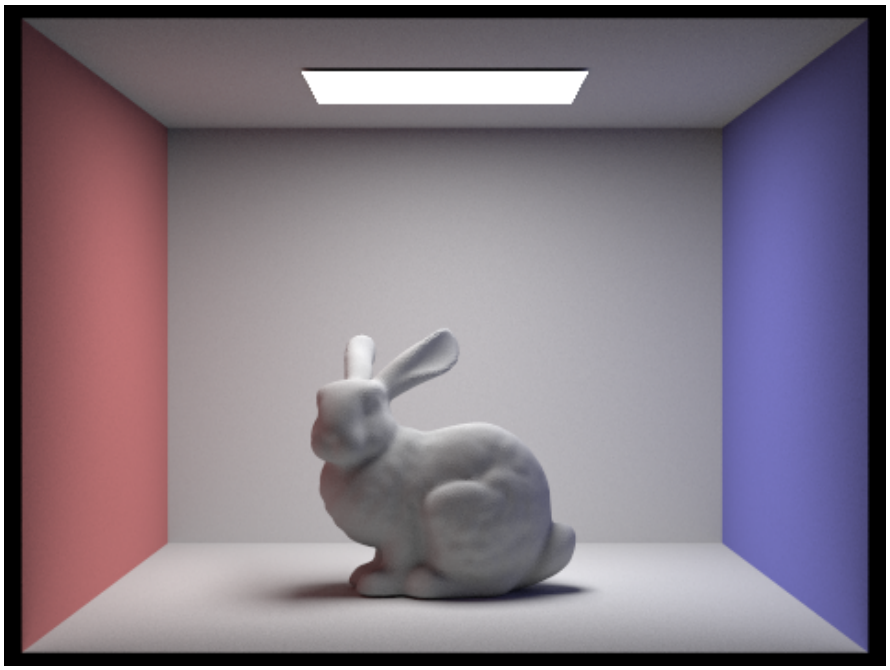
For CBbunny.dae, compare rendered views with max_ray_depth set to 0, 1, 2, 3, 4, and 5 (the -m flag). Use 1024 samples per pixel.



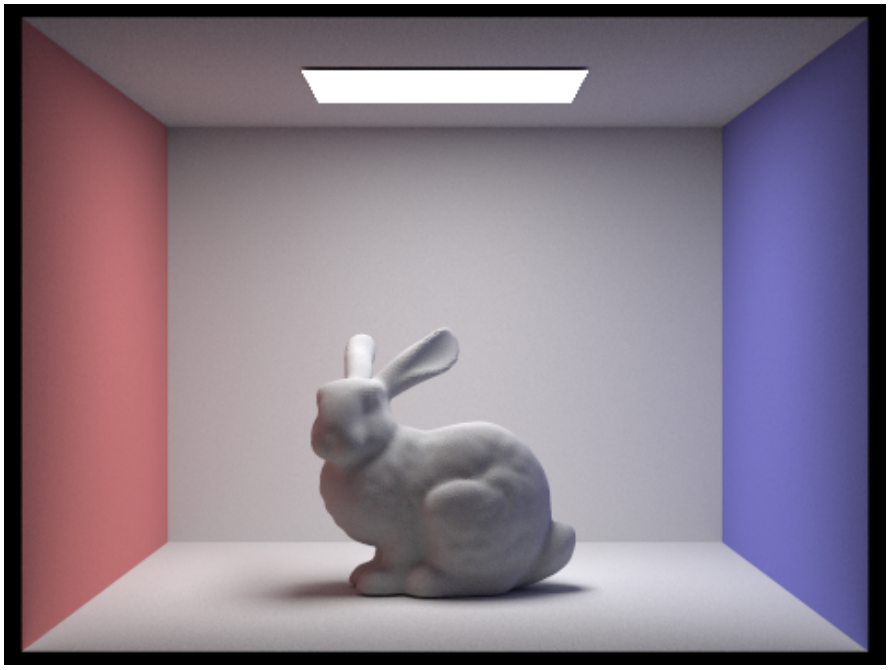
0 max ray depth



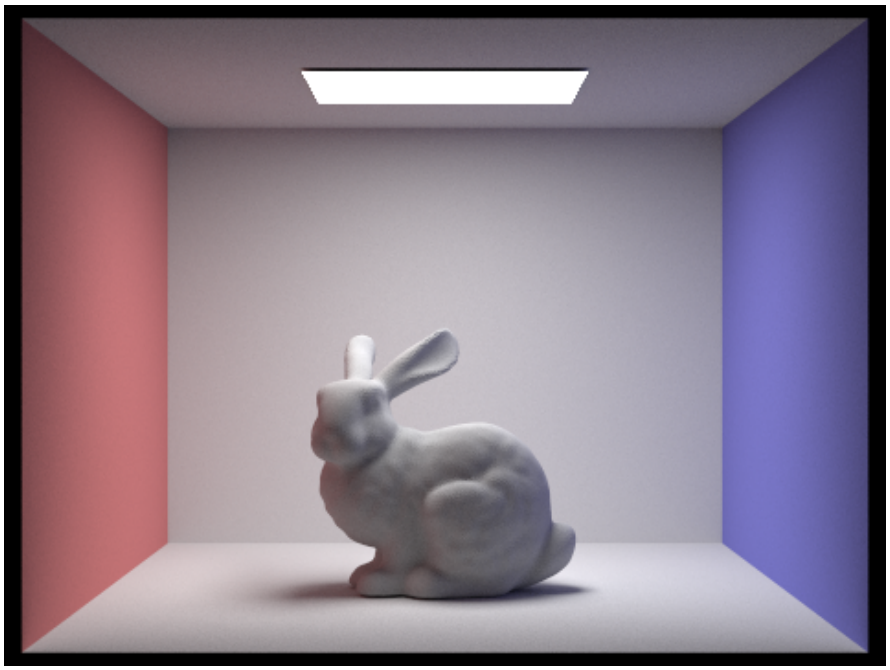
1 max ray depth

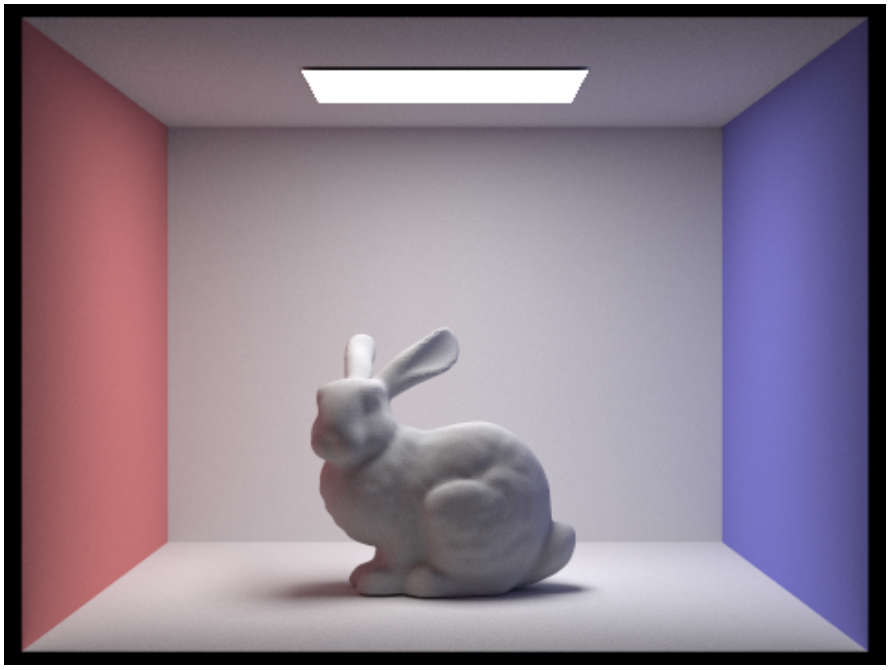


2 max ray depth



3 max ray depth

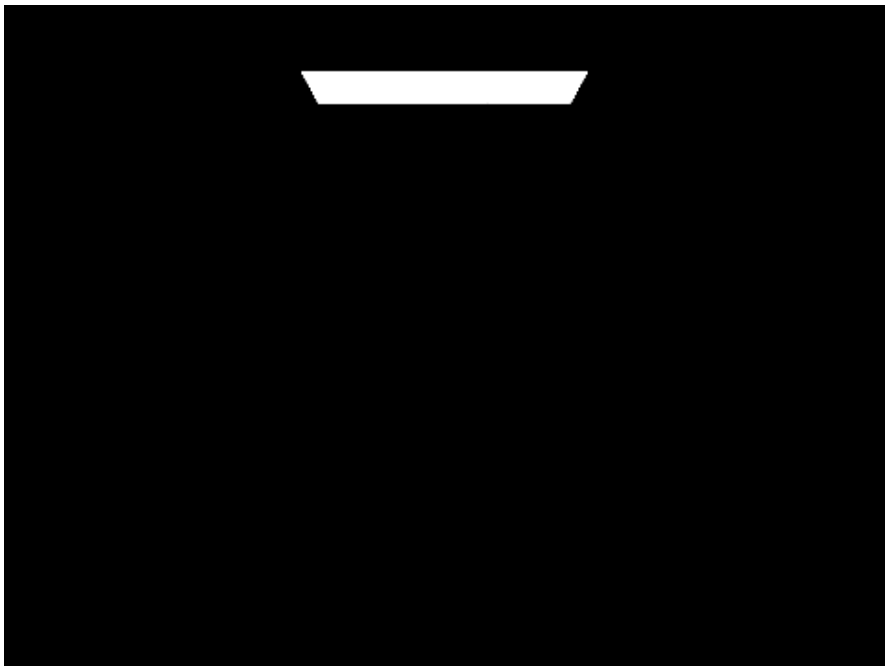
4 max ray depth 

5 max ray depth 

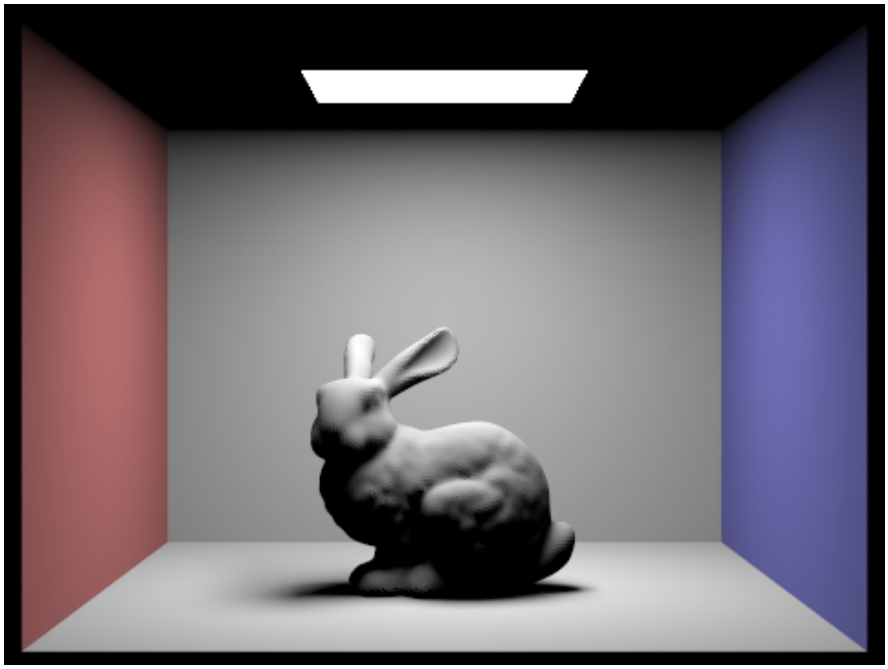
As we increase the max ray depth, the scene is getting brighter (**MORE global illumination**

). This makes sense because the ray depth is increasing, so light is bouncing off of surfaces more times with higher ray depth.

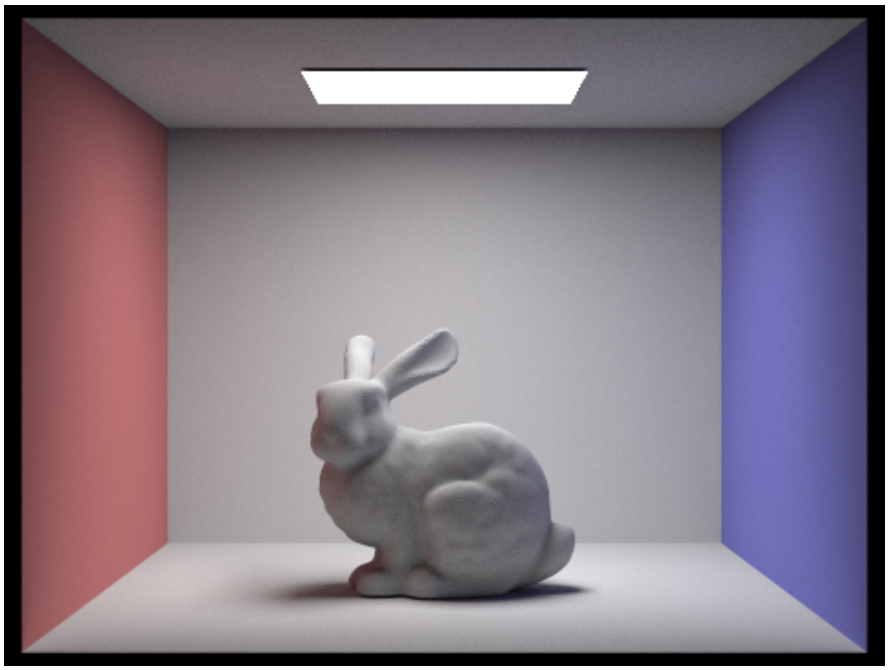
For CBbunny.dae, output the Russian Roulette rendering with max_ray_depth set to 0, 1, 2, 3, 4, and 100



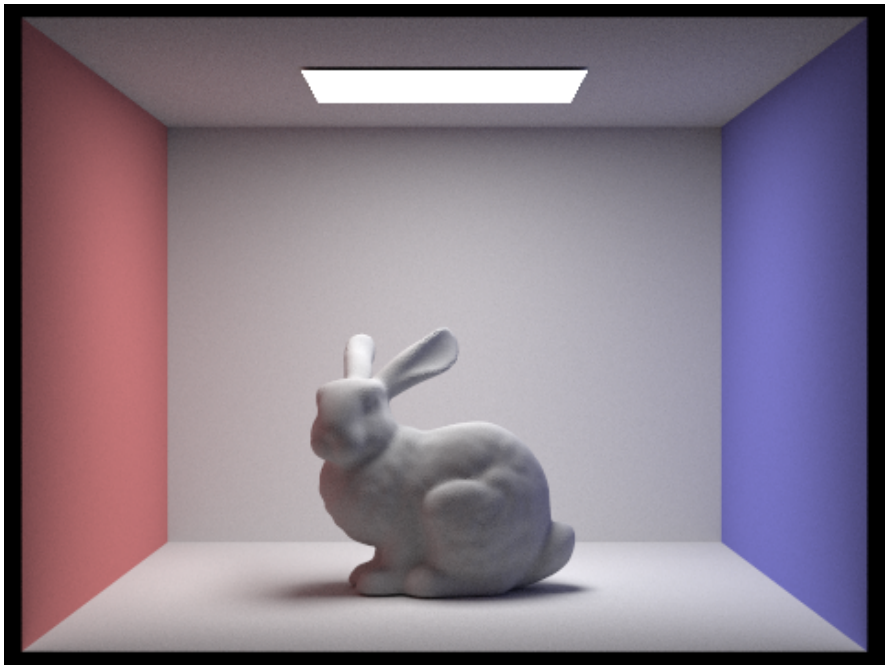
Russian Roulette 0 max ray depth



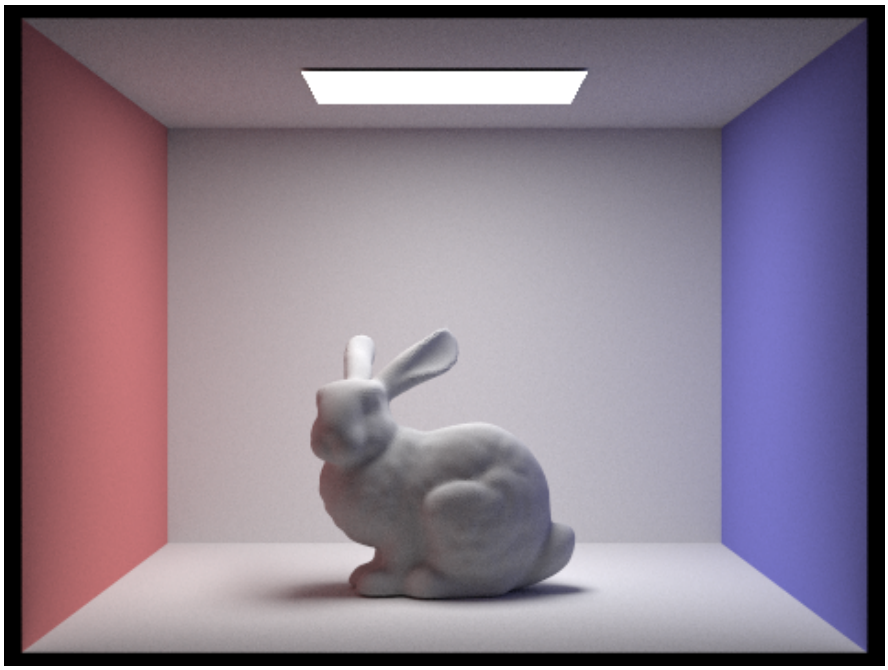
Russian Roulette 1 max ray depth



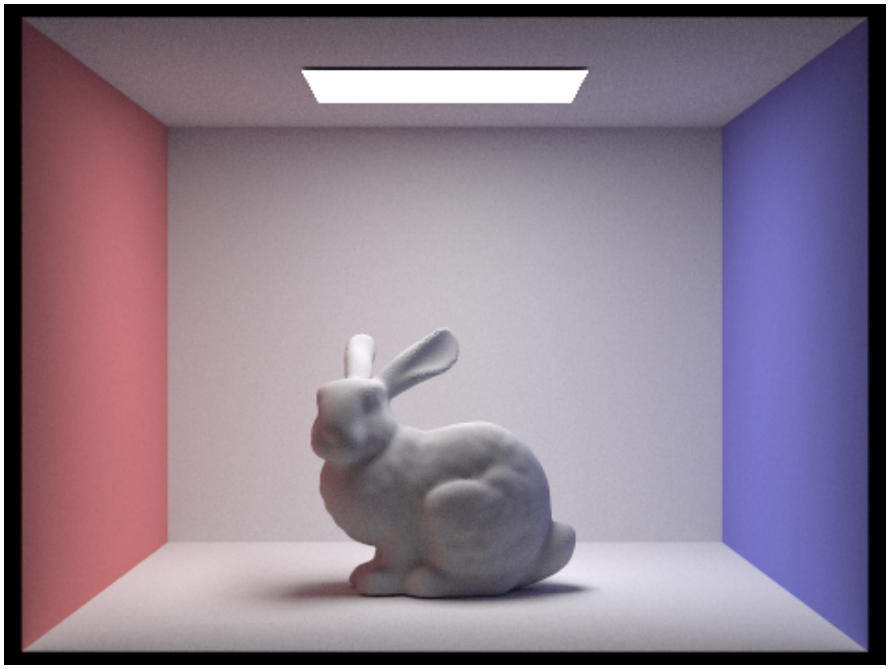
Russian Roulette 2 max ray depth



Russian Roulette 3 max ray depth



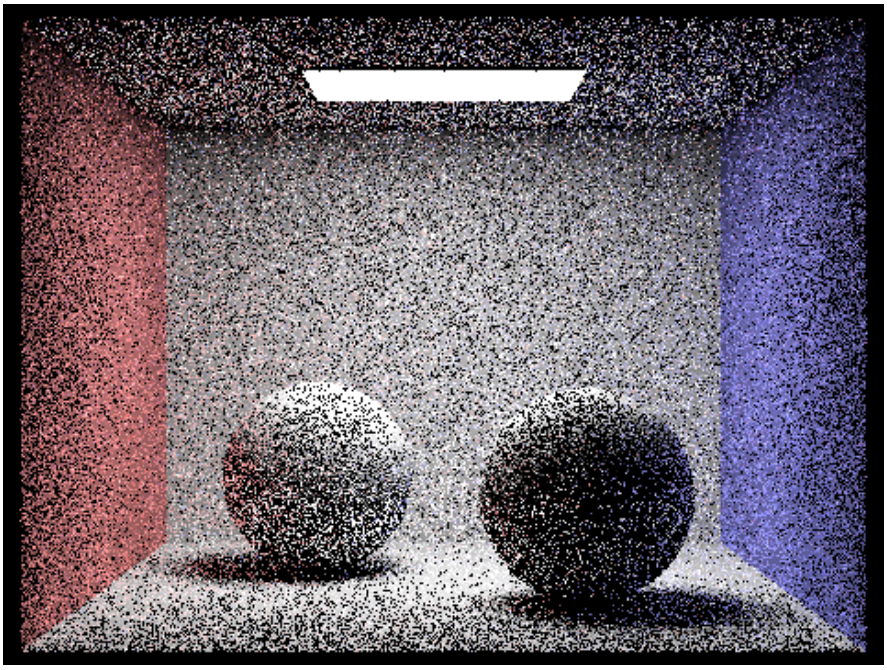
Russian Roulette 4 max ray depth



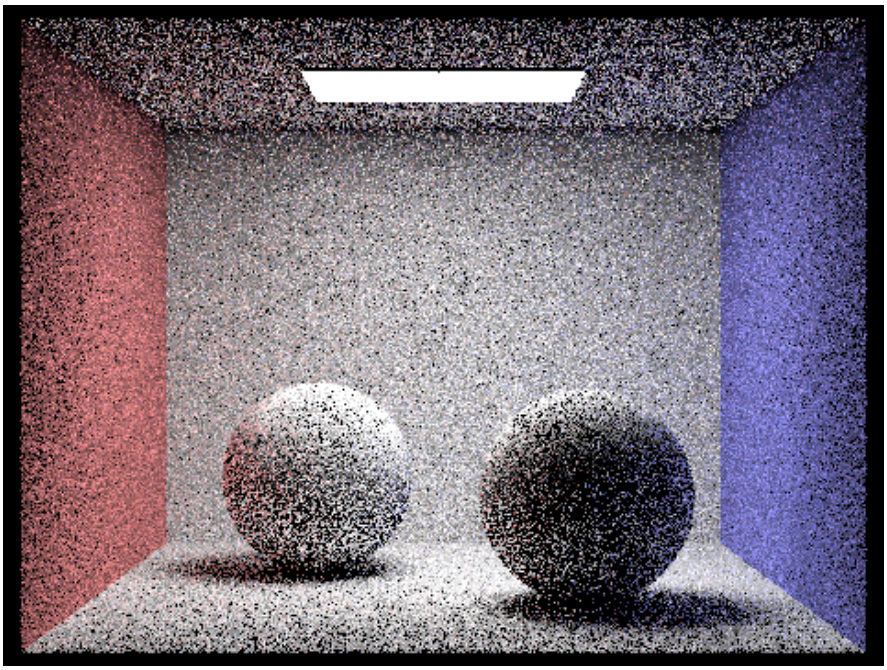
Russian Roulette 100 max ray depth

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.

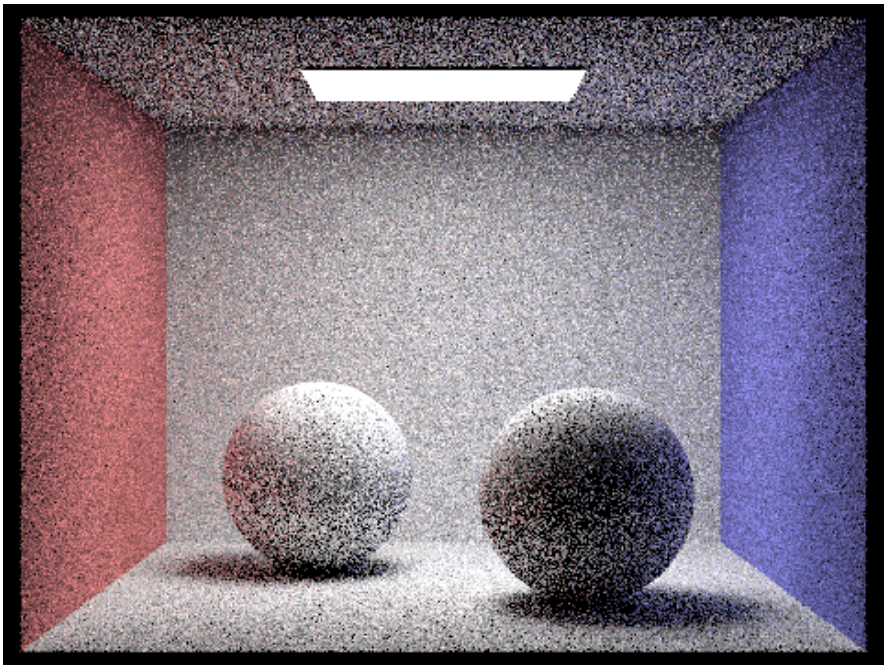
Spheres 0 sample-per-pixel



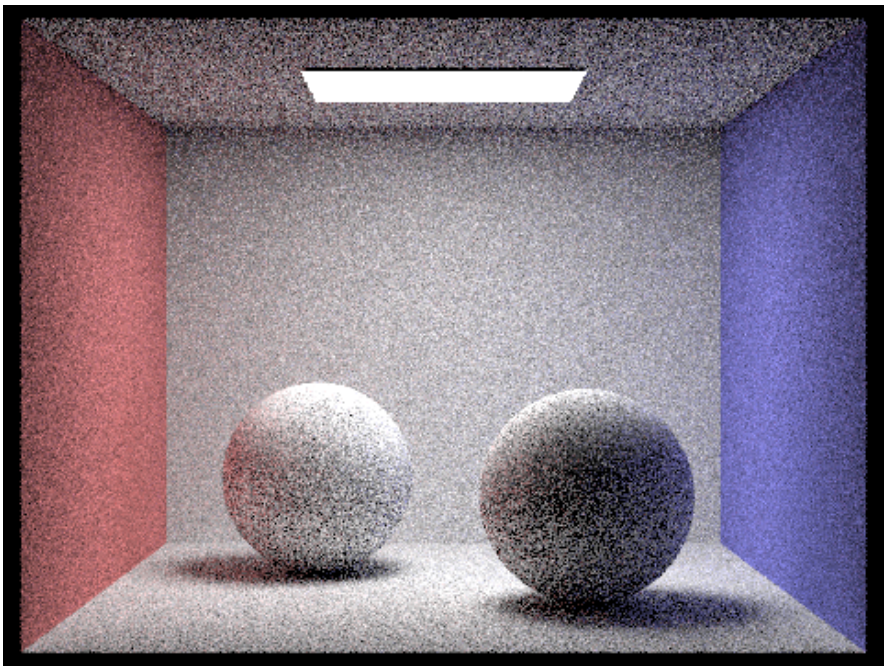
Spheres 1 sample-per-pixel



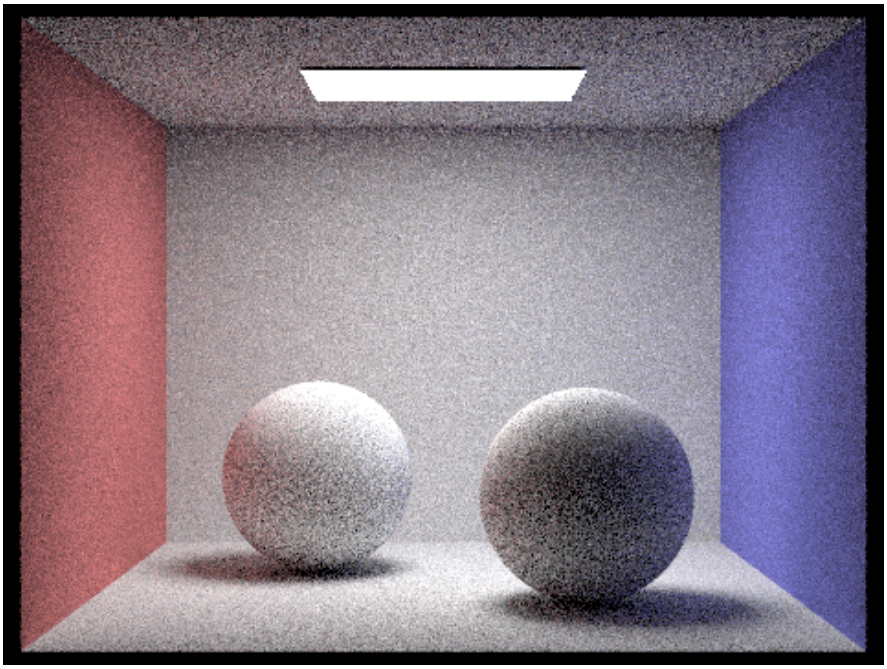
Spheres 2 sample-per-pixel



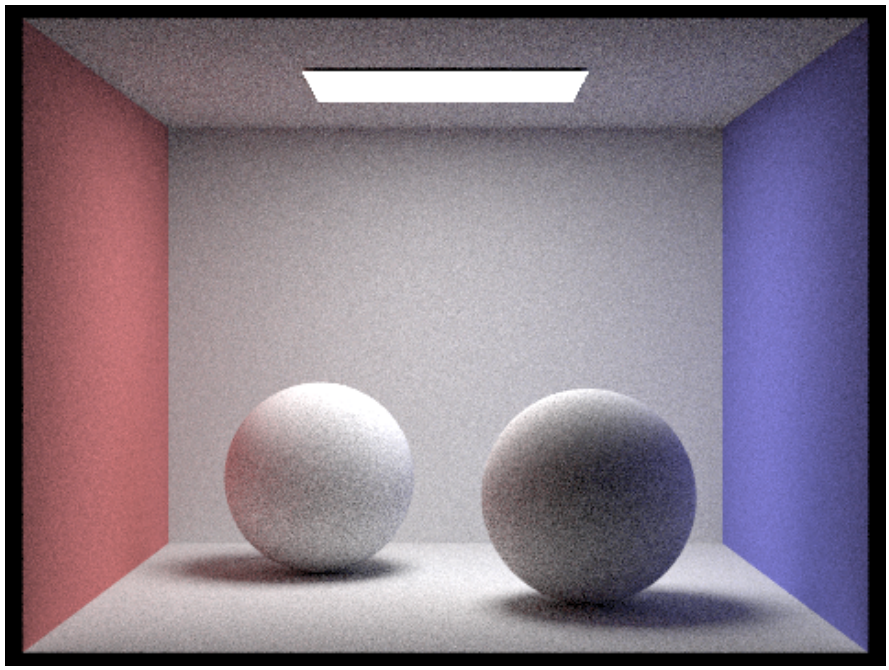
Spheres 4 sample-per-pixel



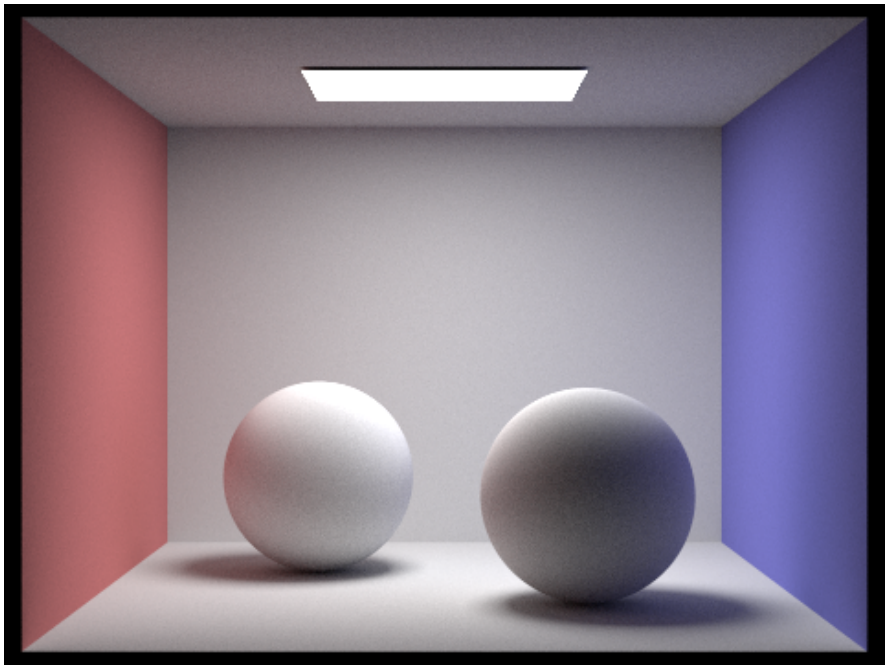
Spheres 8 sample-per-pixel



Spheres 16 sample-per-pixel



Spheres 64 sample-per-pixel



Spheres 1024 sample-per-pixel

Part 5:

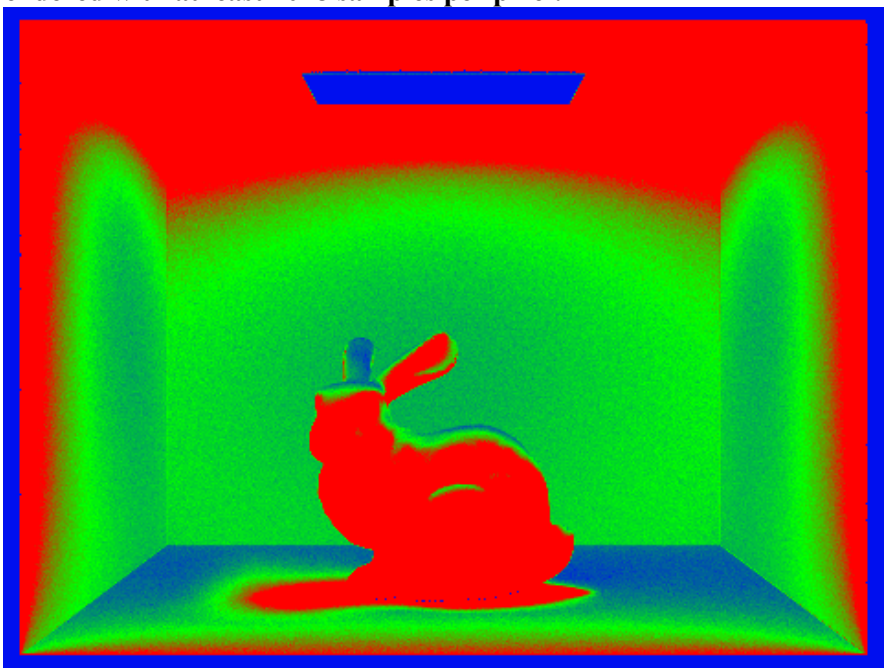
Explain adaptive sampling. Walk through your implementation of the adaptive sampling.

Adaptive sampling is a method used to concentrate samples in harder to render parts of the image, saving computation (high samples) on easier parts of the image to render. Thus, it reduces overall render times by concentrating computate efforts where it's most needed, while still maintaining image quality. In order to implement adaptive sampling, We went through a series of steps. First, as rays are traced through pixels, we calculate statistics of mean (mu) and sum of squares (sigma squared) of the illum values, which we use to measure the variance value, which is an estimate of the noise level of the sample. Then, we calculate I, a confidence interval of the mean illuminance value, which once it converges to a value less than some maxTolerance (param that we set), then we continue, or the max number of samples per pixel are taken. For each loop, we take samples iteratively, and check the convergence. The actual number of samples taken is stored in the sampleBuffer. From there, we use that actual number of samples to calculate the final pixel value. In the end, this dynamically adjusts sampling rate per pixel based on these statistical measures.

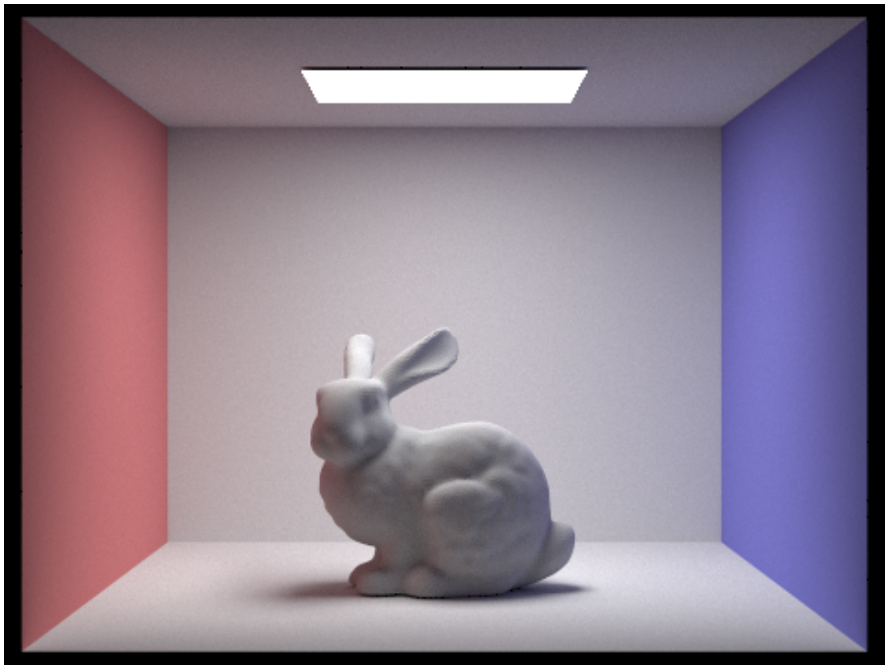
Pick two scenes and render them with at least 2048 samples per pixel. Show a good sampling rate image with clearly visible differences in sampling rate over various regions and pixels. Include both your sample rate image, which shows your how your adaptive sampling changes depending on which part of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

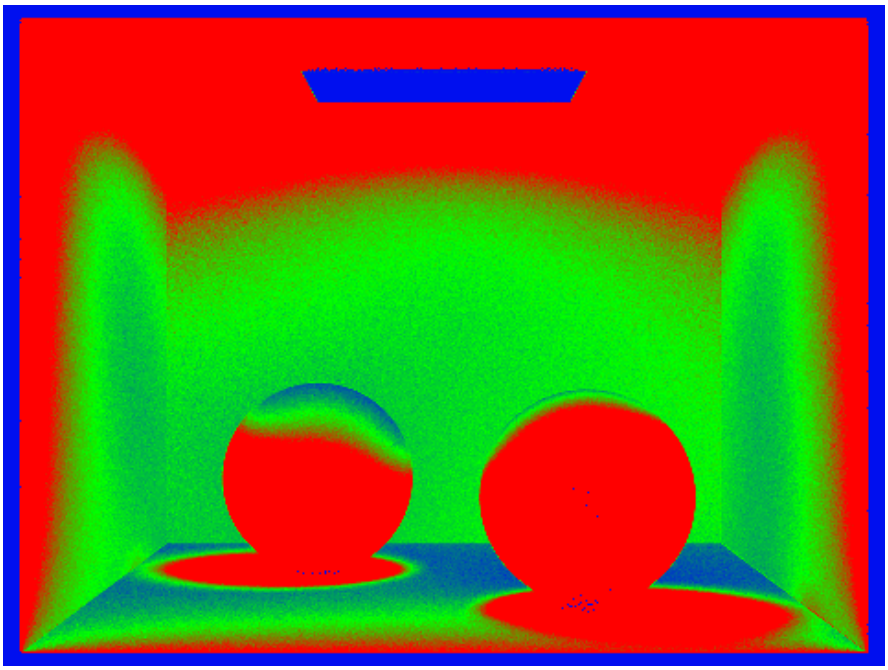
Two scenes rendered with at least 2048 samples per pixel:

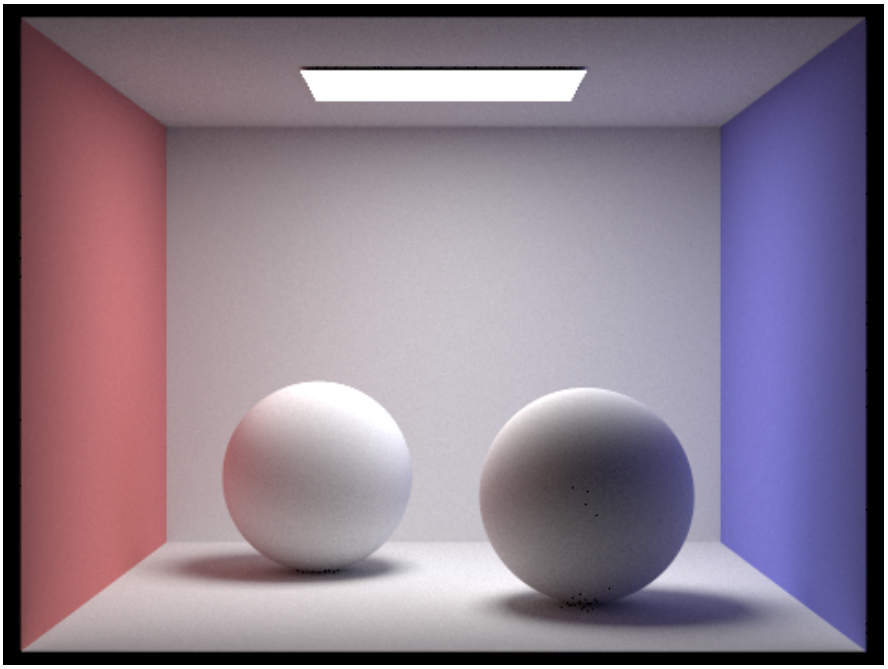
Bunny rate:



Bunny image



Spheres rate: 

Spheres image 

Collaboration Report:

Overall, it was an enjoyable experience collaborating. Joel and Eric met both in-person and over Zoom multiple times through the homework work cycle to discuss conceptual concepts, pair program, and debug. We both utilized Office hours in Soda as well to get questions answered. Joel carried much of the coding sections, as he had a stronger conceptual understanding of the material, while Eric helped in pieces setting up skeleton code based on the spec and debugging. Eric did the majority of the writeup, with the help of Joel to render some of the images. We learned that starting early is very important, and having a plan and understanding the material before coding can save hours of debugging!