Pathtracer

by Jonah Bedouch and Brandon Wong

2025-03-23

This report is also available online at https://cs184-zen.vercel.app/projects/hw03.
The GitHub repository for this project can be found at
https://github.com/cal-cs184-student/sp25-hw3-frustrated/

CS 184 Homework 3: Pathtracer

Overview

In this project, we implemented pathtracting for the purpose of detecting the light values at points in an image from a camera point to generate the final pixel values to be used to generate a resulting image. We first started by implementing ray generation and intersection with the basic primitive objects that make up the different scenes, triangles and spheres. We also implemented the conversion of these lighting values from the samples into pixels. After that, we implemented acceleration via a bounding volume hierarchy due to the ray intersection taking way to long to do if each object had to be checked for each ray to figure out which ones a camera ray intersects. We implemented our own algorithm to find the bounding boxes to obtain a massive speed up in rendering time. After that, we implented direct illumination to find the actual light values at each point in the scenes, both via uniform hemisphere sampling and importance sampling lights. Then we implemented indirect illumination to account for light reflected from points other than the lighting objects. A russian roulette scheme was used to decide whether to bounce rays further or not to obtain an unbiased result and speed up the process a bit. Finally, adaptive sampling was implemented, where sampling only continued at pixels that were still changing enough, ensure not much more samples were taken than needed which helped to significantly speed things up.

Part 1 Ray Generation and Scene Intersetions

Generating Camera Rays

To generate camera rays, the input x and y values are first adjusted by doubling them subtracting one to ensure that the center of the image is at the origin like it will be in camera space. Then, the point on the screen is found by multiplying the resulting offset points by the tan of half the fov of their corresponding direction. This point, along with the origin, can be used to generate a ray in camera space. To then convert this camera space ray into a world space ray, the camere space ray is both rotated by the c2w matrix and translated by the camera position at the same time. Finally, the min_t and max_t are set and the ray vector is normalized. These ray values can be used for primitive intersection by checking if a point within a primitive can be reached

starting from the origin vector and adding the direction vector scaled by a t value. If this t values is between, the min_t and max_t, then the ray is intersecting a primitive.

Generating Pixel Samples

To obtain each pixel, ns_aa samples are generated uniformly within a pixel by using a grid sampler and adding its result to the value of the bottom left corner of each pixel. The scene radiance is then obtained for each sample. The average of all these samples are used for the overall scene radiance of the pixel, which is updated with that value.

Ray-Triangle Intersection

We used the Moller Trumbore algorithm to calculate triangle intersection. To implement this, we found a few variables:

- The edges E_1 and E_2, which represent the vectors pointing away from point p_1 at points p_2 and p_3 respectively
- The vector S, which is a vector pointing from p_1 at the origin of the ray.
- The vector S_1, which is the cross product between the ray direction and E_2 (representing the normal vector of the plane that both the ray and E_2 lie on)
- The vector S_2, which is the cross product between S and E_1 (representing the normal vector of the plane both the vector pointing back at the camera and E_1 lie on)

These values come together to help us find t, b_1, and b_2 (where b_1 and b_2 are our barycentric coordinates). This happens using the following algorithm. First we take the dot product between S_1 and E_1, which represents the projection of the normal of the plane between the ray direction and our second edge onto our first edge. In other words, this represents the point along our first edge that is closest to the normal between the ray direction and the second edge, which helps us understand the orientation of the triangle relative to the ray, and also allows us to normalize the values to be on the correct scale. From here, we can use this dot product to find 3 values:

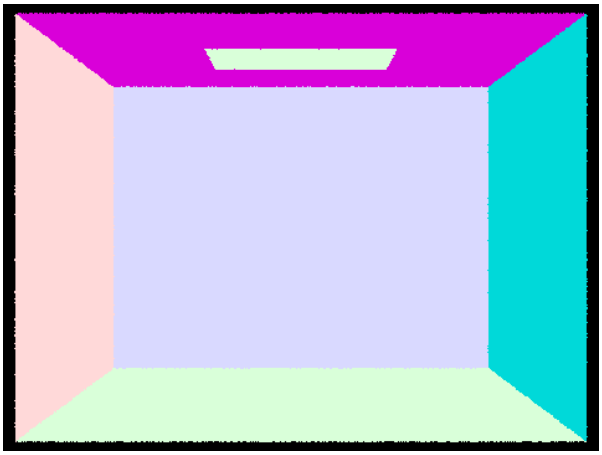
- t, the distance along the ray to the intersection point, which is found as $S_2{*}E_2$ divided by the afformentioned dot product.
- b_1, the barycentric coordinate u, which is found as $S_1{*}S$ divided by the afformentioned dot product.
- b_2, the barycentric coordinate v, which is found as $S_2{*}r.d$ divided by the afformentioned dot product.

Given a set of barycentric coordinates, we can determine whether the intersection point is contained within the triangle dependent on whether both b_1 and b_2 are positive numbers whose sum does not exceed 1. If we find that the point is within the triangle, we find the interpolation between the normals by using these barycentric coordinates (where N_1 is multipled by $1 - b_1 - b_2$, N_2 is multiplied by b_1, and N_3 is multiplied by b_2).

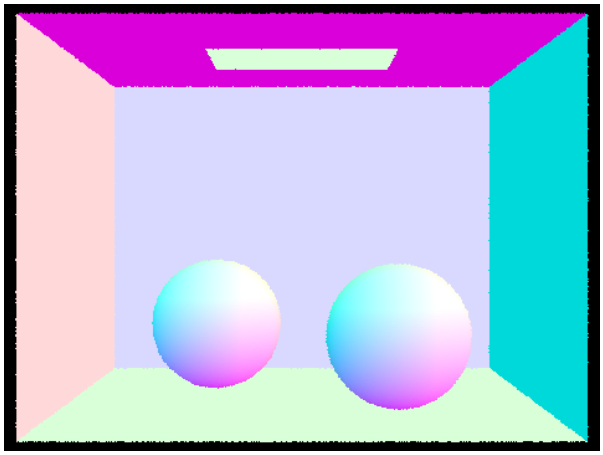
Task Ray-Sphere Intersection

To check for an intersection along a sphere, a parametric equation was used and solved for. Because the intersection point along a sphere minus the center squared results in the same values as the radius squared, and a point along a sphere would just be the origin plus the vector from the origin (the camera point) to the point which is the ray vector times a value t, a parametric equation in the form $at^2 + bt + c = 0$ can be developed where a is the square of the ray vector, b isthe dot product of twice the value of the origin minus the center and he ray vector, and c is the square of the origin minus the center with the square of the radius then subtracted from that. Once those values are found, the parametric equation can be solved for t to find the two intersection point on the sphere. As long as one of the points are above 0, the ray intersects the sphere.

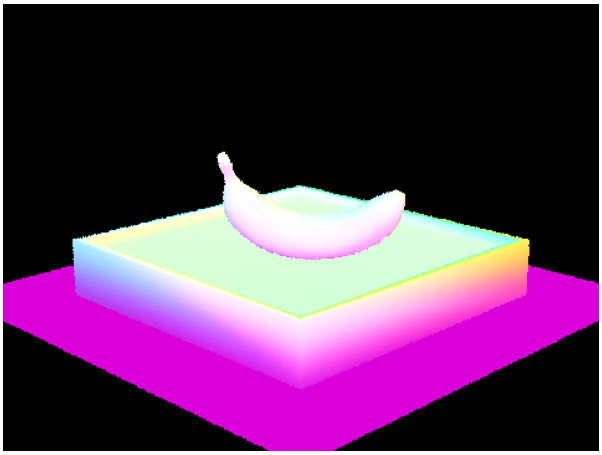
Once it has been shown that the ray intersects the sphere, the intersection values need to be updated. The t and max_t values are updated with the smallest t value above 0, and the surface normal is the vector of intersection minus the camera point, normalized. The primitive is the sphere itself and the bsdf is obtained with get_bsdf.



Part 1 CBempty



Part 1 CBspheres



Part 1 banana

Part 1 teapot

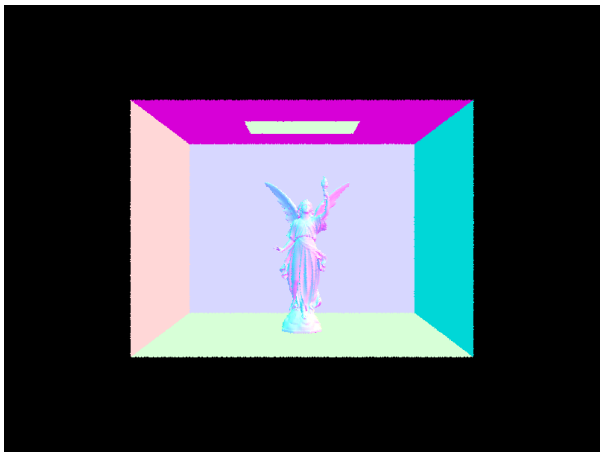
Part 2: Bounding Volume Hierarchy

BVH Construction Algorithm

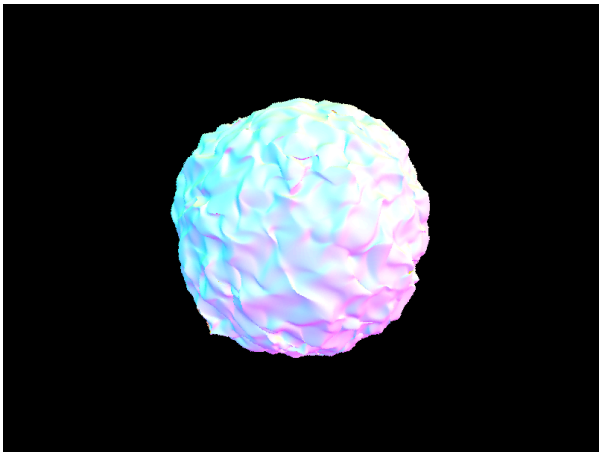
To decide where to split the bounding boxes when constructing our BVH for each node, we started by finding averge of the centroid for each primitive within the current bounds. Then, after making sure that the number of primitives in the bounding box is larger than the maximum leaf size, we found the number of objects to one side of the centroid along each axis. Whichever split along each axis that had the closest number of objects on each side, which was checked bby taking the absolute value of the number of objects on one side minus half the total number of objects, was the axis we decided to split on. We then sorted the objects within the range of the current node such that the primitives on one half of the centroid axis we decided on were on the left and the rest were on the right. Then, we recursively constructed the left and right nodes, where the two nodes were split by taking the starting point of the parent node and adding the number of objects on the left side to it to find the point where objects begun to be on the other side in the sorted list. Finally, the current node was returned.

BVH Acceleration Dependent Images

Part 2 Cow



Part 2 CBlucy



Part 2 Blob

Part 2 Maxplanck

Rendering Time Analysis

The cow.dae file took approximately 5 seconds to render without the implementation of BVH acceleration, and 0.04 seconds after BVH acceleration. For banana.dae, it took around 2 seconds to finish before and 0.02 seconds after. For maxplanck.dae, it took around 55 seconds before and 0.11 seconds after BVH acceleration was implemented. The blob.dae file took around 300 second to render before and 0.22 seconds to render after. For the simpler images, BVH acceleration already had a massive increase in rendering speeds, with the speed up resulting in render times around ten times as fast as before. For larger images, this speed up was even more pronounced, with the maxplanck file obtaining around a 500 times speedup and the even larger blob file reaching around a 1500 times speed up from before.

Part 3: Direct Illumination

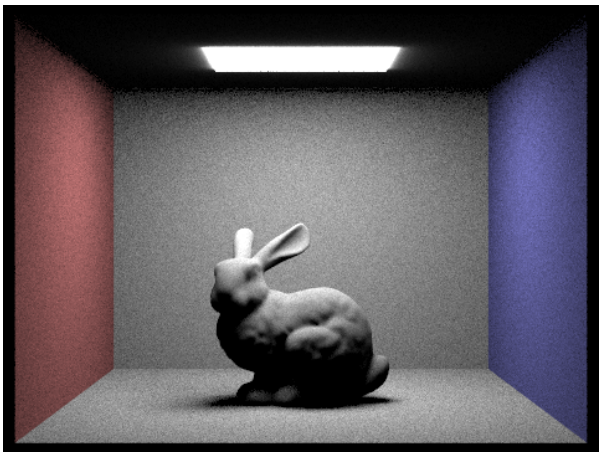
Direct Lighting with Uniform Hemisphere Sampling

Direct lighting via uniform hemisphere sampling works by getting a normalized direction vector in object space, using it to create a new (world space) ray, and then checking if this ray intersects a light source. If it does, the reflection equation, shown below, can be used to calculate the illuminance which is then added to an overall output illuminance variable. This is done for a total number of times equivalent to the total number of samples to be taken for each point. This overall value is then normalized by the total number of samples and returned as the direct lighting for that point.

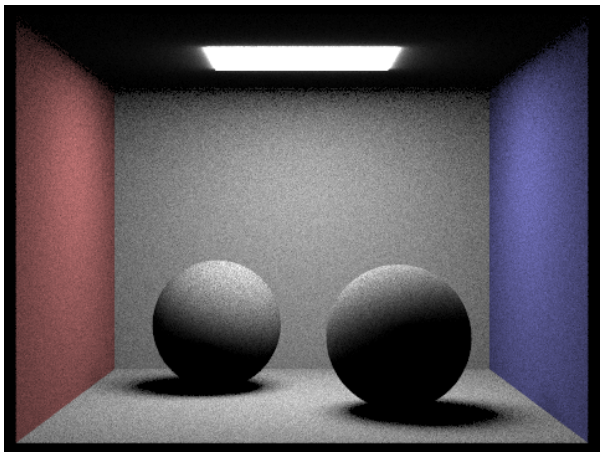
To calculate the reflection equation, the light emitted from the light source was multiplied by the material reflectance from the angle and the cosine of the angle and divided by the probability of that sampled incoming angle being taken, which for uniform hemisphere sampling is just 1 divided by two time pi due to each direction having the same sample probablility. The cosine of the angle was calculated by taking the dot product of the normal of the intersection point and the sample angle in world space, which is just the z value of the sample angle due to how object space is set up.

$$\frac{1}{N}\sum_{j=1}^{N}\frac{f_r(\mathrm{p},\omega_j\rightarrow\omega_r)\,L_i(\mathrm{p},\omega_j)\,\cos\theta_j}{p(\omega_j)}$$

Reflection Equation



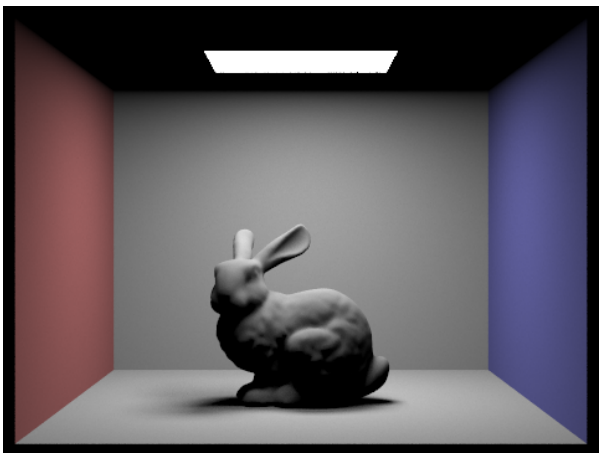
Part 3 CBbunny Hemisphere



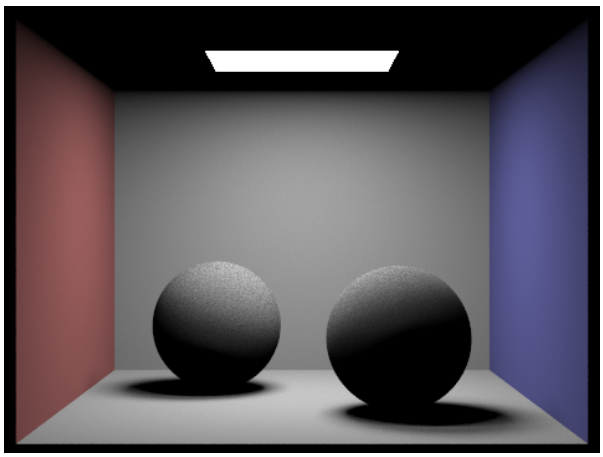
Part 3 CBspheres Hemisphere

Direct Lighting with Importance Sampling Lights

Direct lighting via importance sampling lights works by going through each light and finding a direction from a point on the light to the current point intersected by the camera ray. After that, a new ray from the light to the point is created, and then it is checked if there are any objects in the way by seeing if there is an intersection with a closer t value than the point intersected by the original camera ray. If there isn't the reflectance equation is used to find the overall output light. This is done a total number of times equivalent to the total number of samples if the light source is not a point, and only done once if it is a point source, as a point source only has one point to check light rays from while a regular light source has points all across itself whose overall average light to a point needs to be checked.



Part 3 CBbunny Importance



Part 3 CBspheres Importance

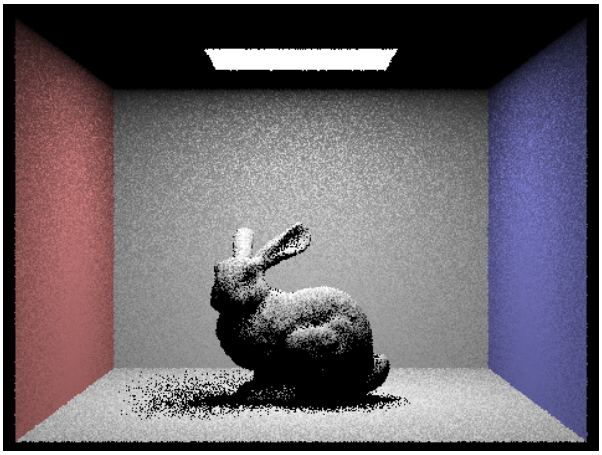
Comparison Analysis

Hemisphere sampling has results that are distinctly fuzzier than importance sampling. This can be seen in both previous images. This is likely because hemisphere sampling depends on the random chance of a sample ray hitting the light source, and even that can happen more or less

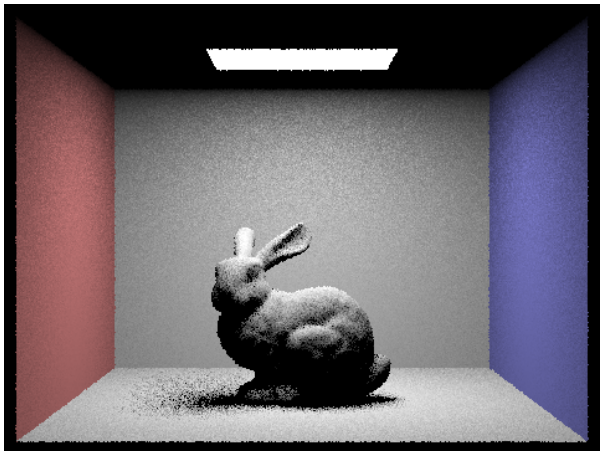
often than an average so the exact value for close by points that logically should have very similar values can be more off than they should be. This change across points from random chance is likely what causes the fuzziness, and because importance sampling doesn't depend on luckily hitting the light source it results in much smoother and consistent images across each point. It is likely that hemisphere would get smoother with more samples as each point would end up closer to the actual average they should be.

Shadow Results on CBbunny

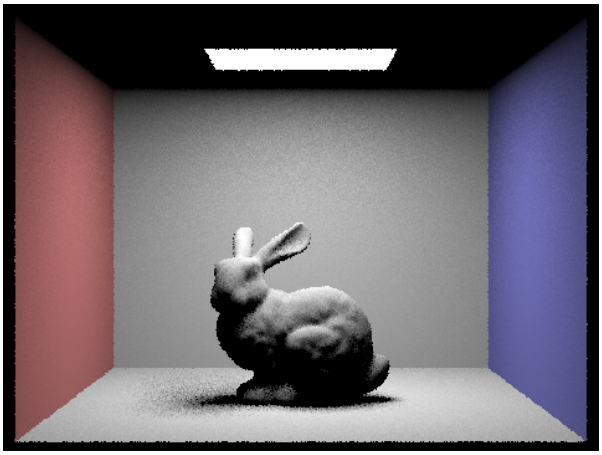
As the number of light rays increases, the shadows improve in smoothness and consitency. Because the light source can be more than a point, some points may result in a shadow and others may not. This results in a random chance that a point partially within view of a light may not recieve a light ray at lower numbers of light rays, or may not receive a proportion of light rays equivalent to the proportion of the light visible from that point, resulting in more uneven and fuzzy shadows. With more light rays, the proportion of rays reaching the point from the light becomes closer to what is should be, resulting in better shadows.



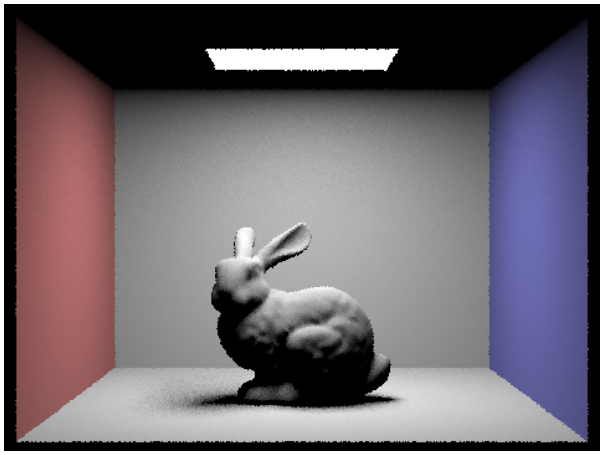
Part 3 CBbunny 1 Light Ray



Part 3 CBspheres 4 Light Rays



Part 3 CBbunny 16 Light Rays



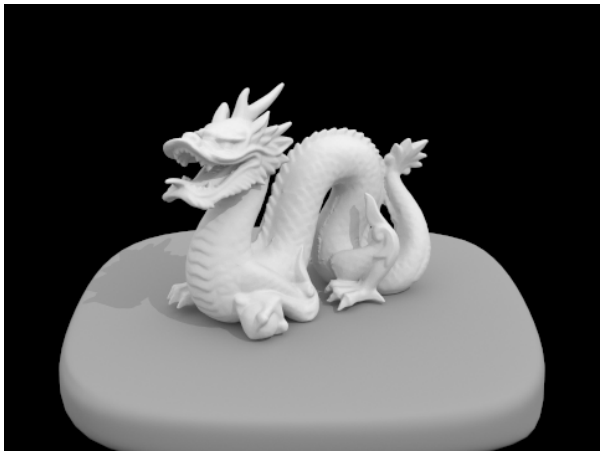
Part 3 CBspheres 64 Light Rays

Part 4: Global Illumination

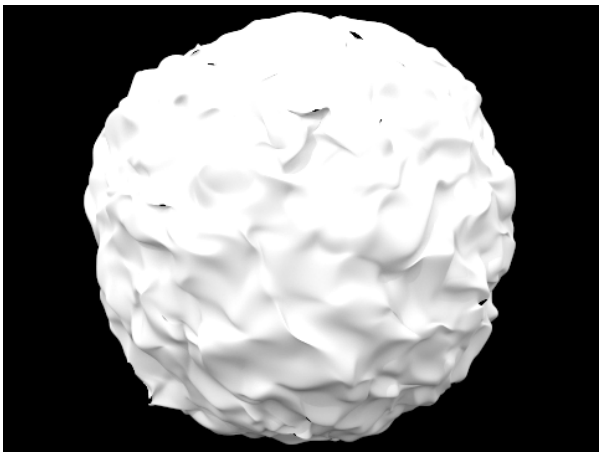
Indirect Lighting Implementation

When the indirect lighting function is first called, it first checks if the ray depth is greater than zero. If it isn't, it immediately returns the zero vector. Once past that first check, it takes a sample in a random direction to be the direction the light is coming from, returning the material reflectance of the angle, the angle itself, and the probability of selecting that angle. This is used to form a new ray, starting from the intersection point of the camera ray and going in the direction of the incoming light ray. Then, the function decides whether or not to add the direct lighting to the overall incoming light. If the pathracer is set to accumulate bounces or if it's the final bounce, then the direct lighting is added to the current result. Next, the termination probability is decided on. If it's the first bounce, the termination probability is set to 0 to gurantee at least one bounce. Otherwise, the termination probability is set to 0.3. Finally, if it's decided not to terminate the ray and the new ray intersects an object, a lighting value is obtained by first recursively calling the indirect lighting function and taking the resulting value and multiplying it by the angle material reflectance and the cosine of the angle between the normal and the direction of the incoming light and dividing that by the probability of taking that direction for the incoming light as well as the probability of continuing (1 minus the termination probability). This lighting value is added to the overall lighting value, and this final value is returned to calculate the overall light at a point, including both the direct and indirect lighting.

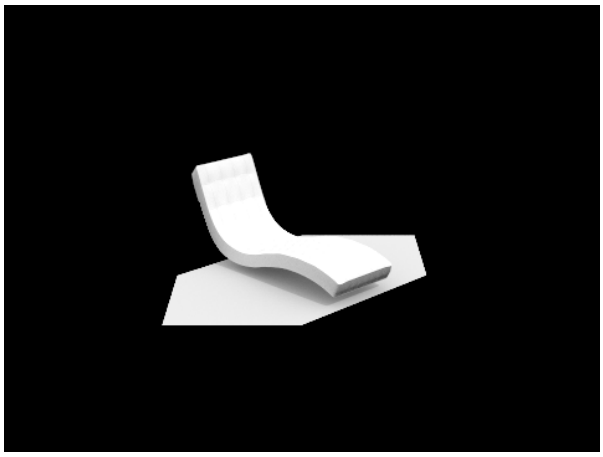
Part 4 Bunny



Part 4 Dragon



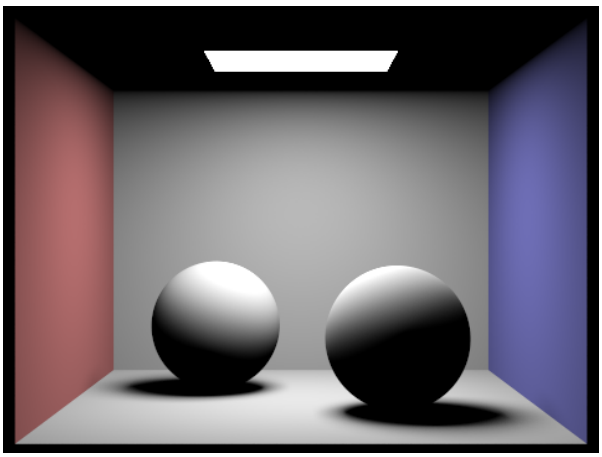
Part 4 Blob



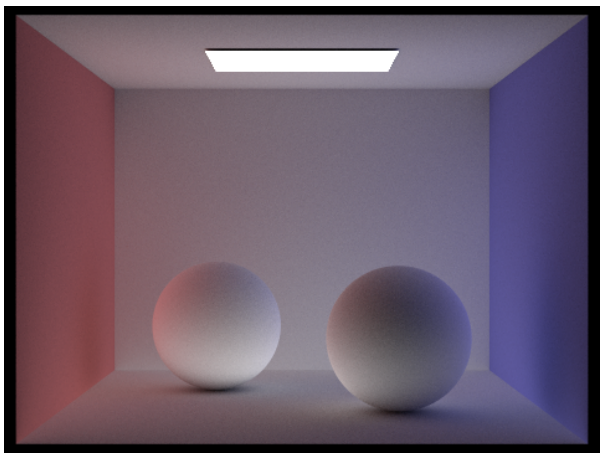
Part 4 Bench

Direct Illumination vs Indirect Illumination

The direct illumication only covers the tops of the spheres facing the light and the walls. The tops of the walls and the sides of the balls get dark very quickly, and the ceiling and bottoms of the spheres themselves are pitch black as no direct light can reach them. The results of just indirect illumination are much more even with light being visible everywhere. The brightest parts are actually the bottoms of the spheres on the sides of them where the light is, which makes sense as a lot of light would be reflecting off the floor there. The side of the closer sphere and the parts of the box at the edges are the darkest, which makes sense as there is no surface for light to reflect on to them. The overall light level of the indirect lighting image is lower than direct lighting, making sense as redlected light doesn't keep the full brightness of the original light unless the material is very reflective.



Part 4 CBspheres Direct

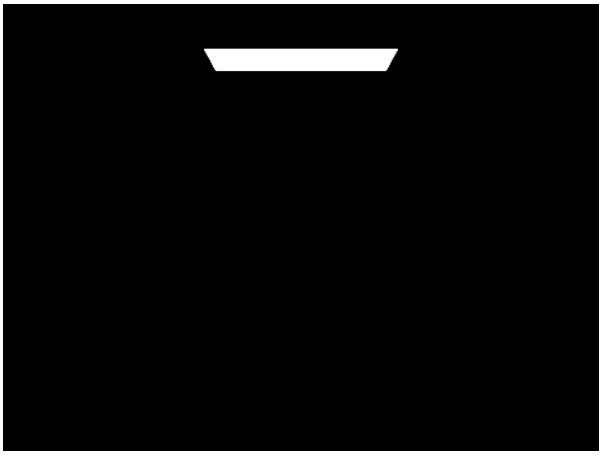


Part 4 CBspheres Indirect

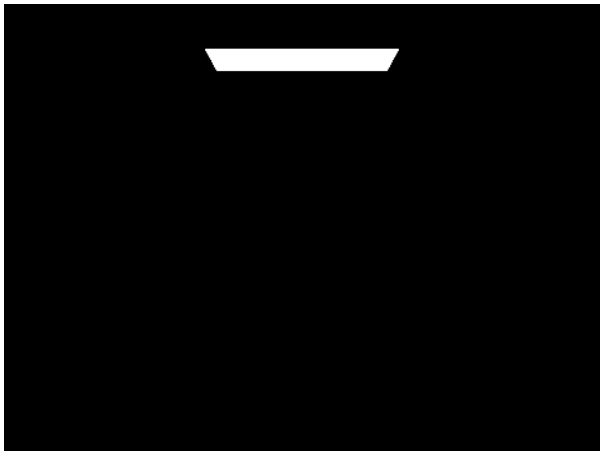
CBbunny Accumulated vs Unaccumulated Bounces

While the first bounce of the light is just the direct lighting, the second and third bounces help light up the rest of the image and even add some color to the reflections. To be specifc the second

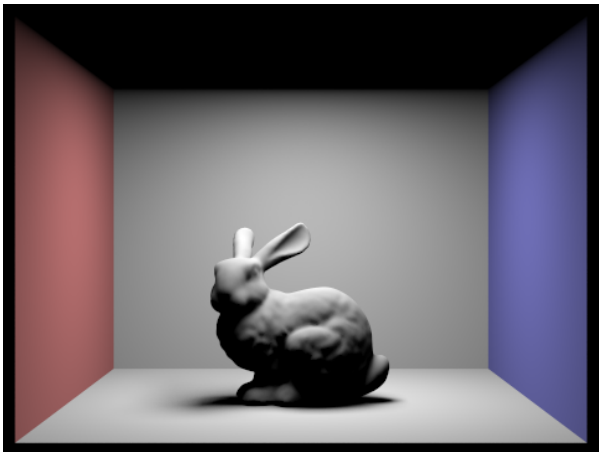
bounce adds an even level of light everywhere in the box, coming directly from the reflections of the first bounce to every point in the image. The third bounce adds some more color. While the second bounce adds some color in the reflections itself, the third bounce has slightly more distinct colors in its reflections that improve this slight addition of color to the normally white walls even more, helping the second bounce to get past the pure white colors of the white parts of the first image.



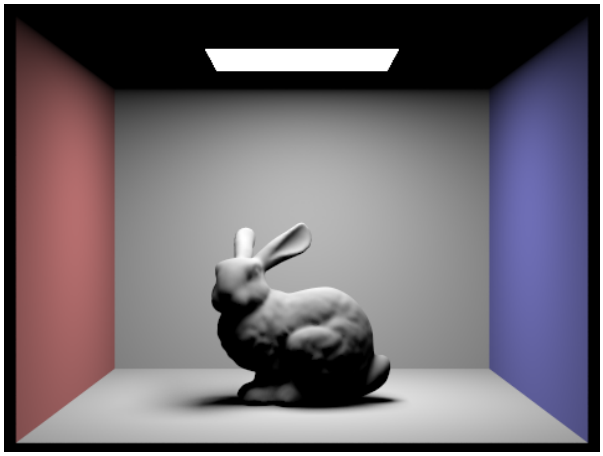
Part 4 Unaccumulated Bounce 0



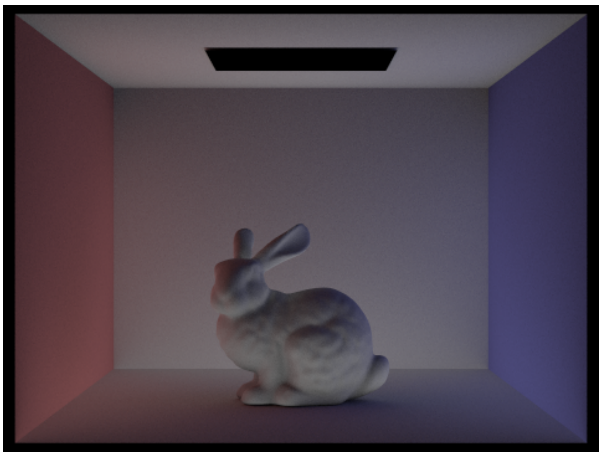
Part 4 Accumulated Bounce 0



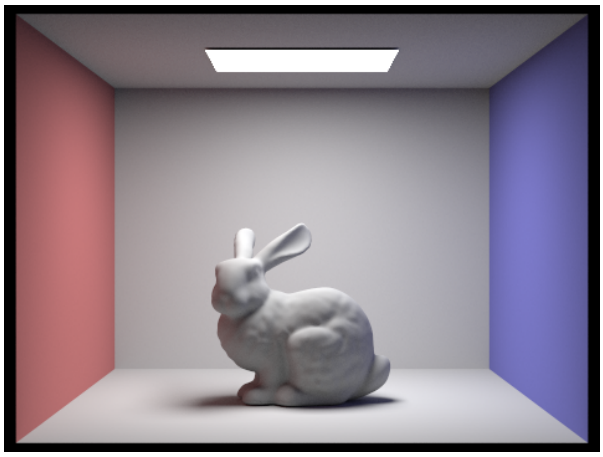
Part 4 Unaccumulated Bounce 1



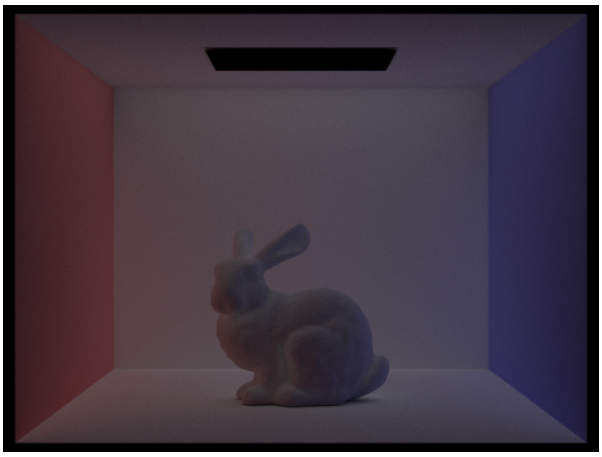
Part 4 Accumulated Bounce 1



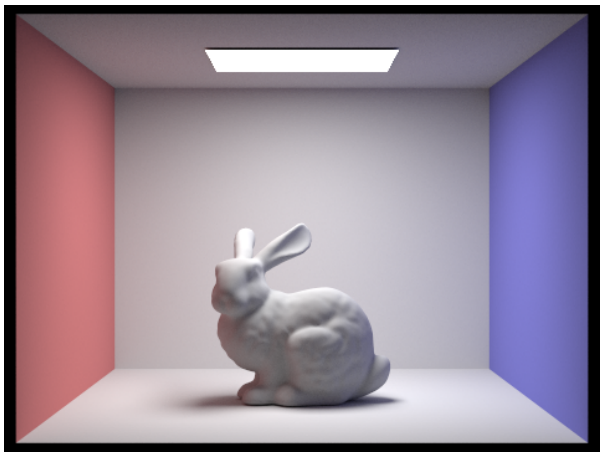
Part 4 Unaccumulated Bounce 2



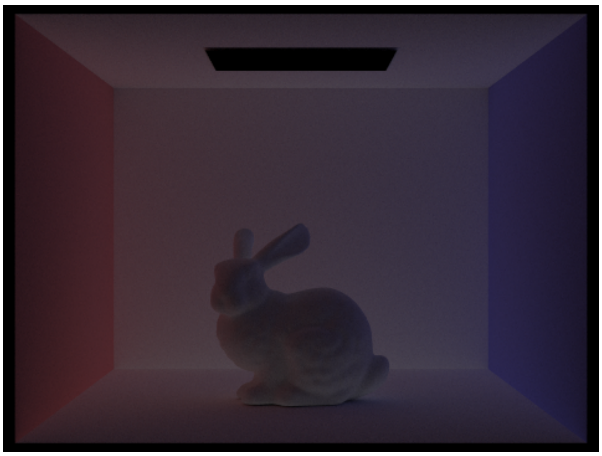
Part 4 Accumulated Bounce 2



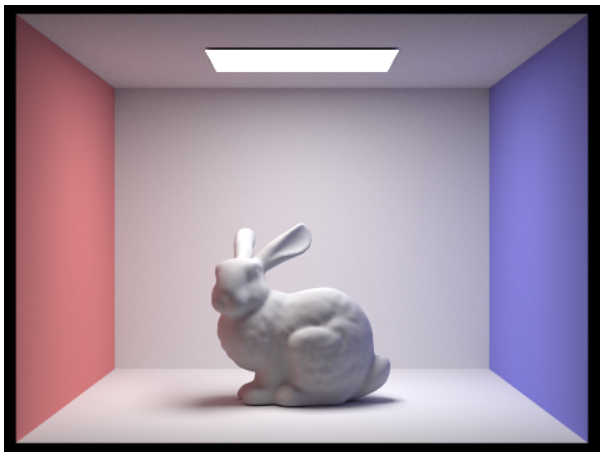
Part 4 Unaccumulated Bounce 3



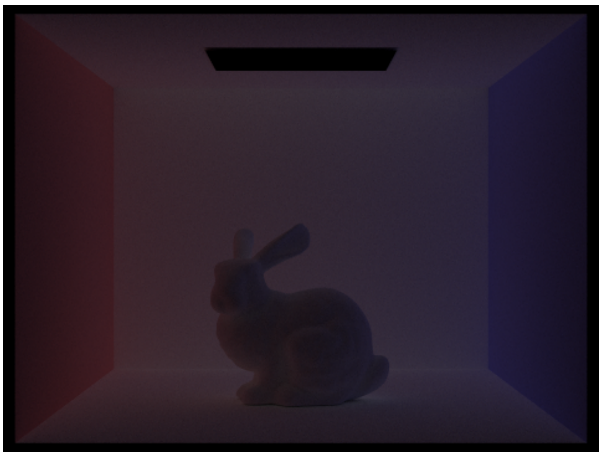
Part 4 Accumulated Bounce 3



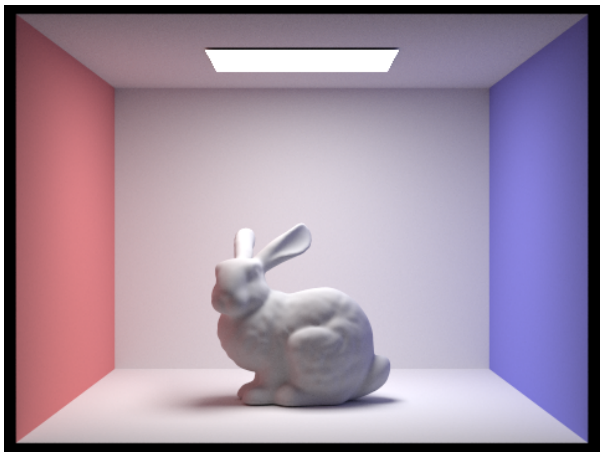
Part 4 Unaccumulated Bounce 4



Part 4 Accumulated Bounce 4

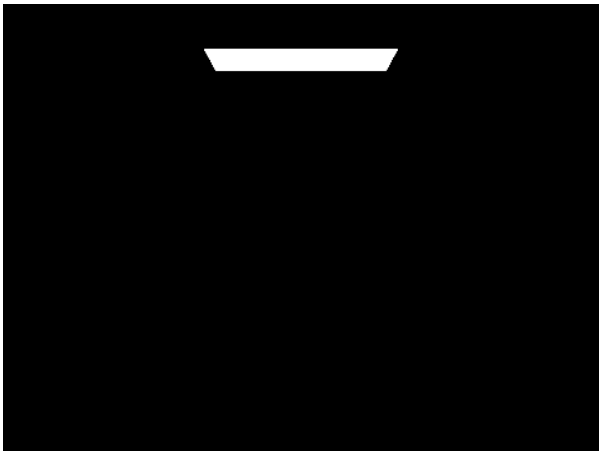


Part 4 Unaccumulated Bounce 5

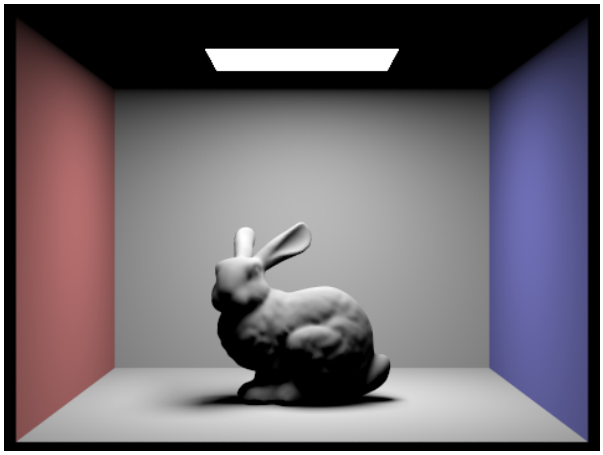


Part 4 Accumulated Bounce 5

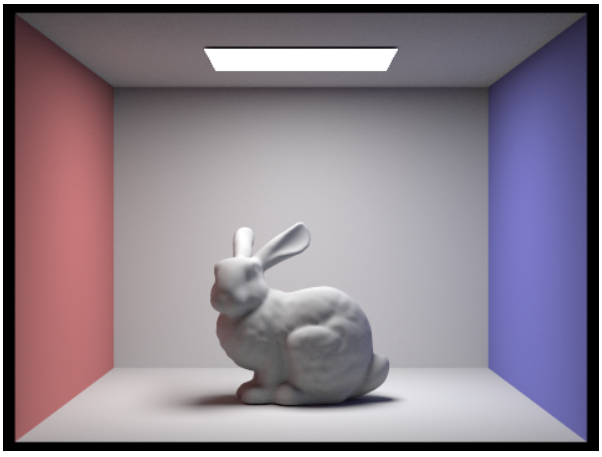
Russion Roulette Results



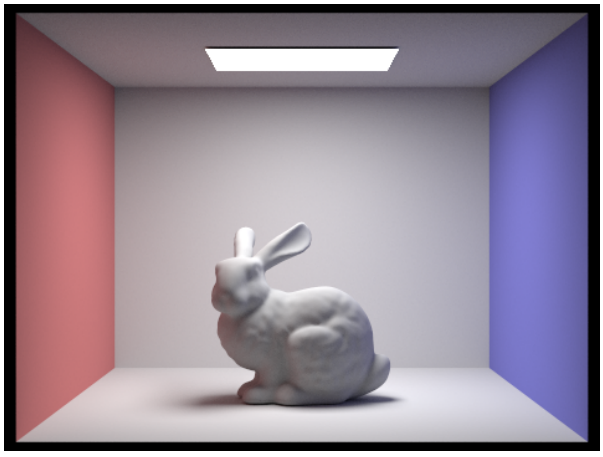
Part 4 RR Depth 0



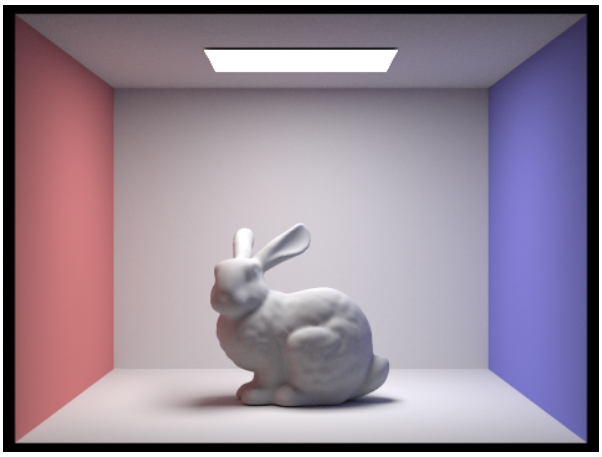
Part 4 RR Depth 0



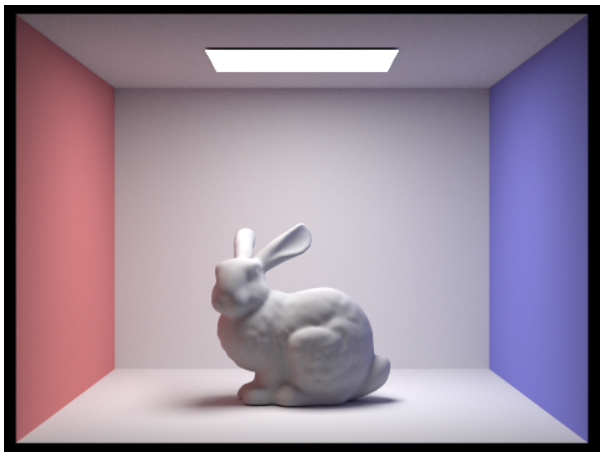
Part 4 RR Depth 2



Part 4 RR Depth 3



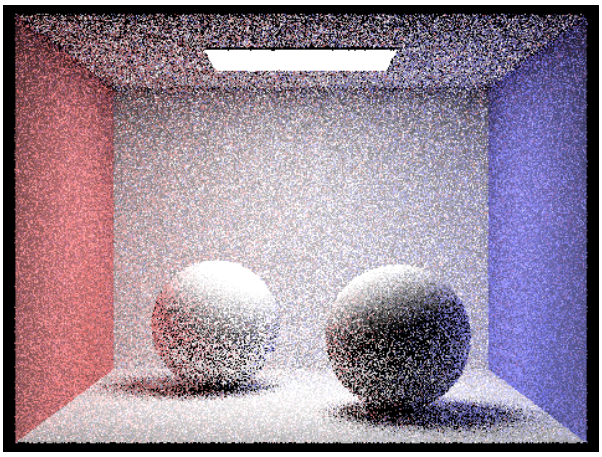
Part 4 RR Depth 4



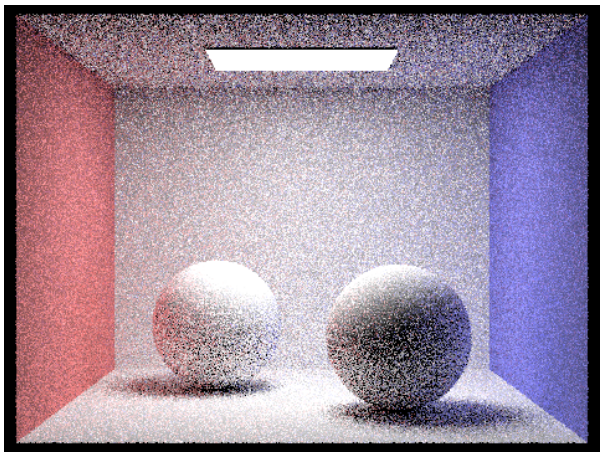
Part 4 RR Depth 100

Different Sample per Pixel Rates

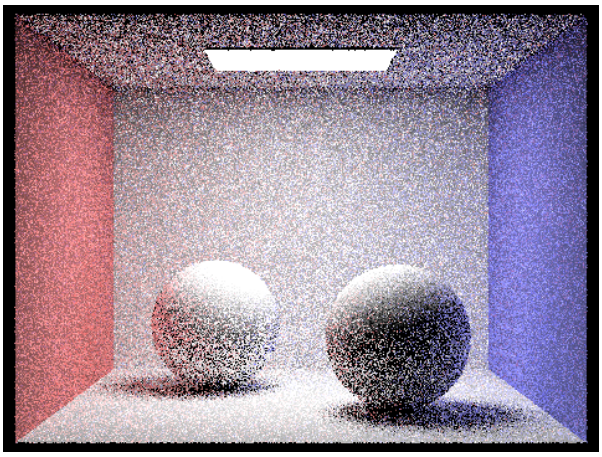
Lower sample rates have a lot of fuzz made up of very bright pixels fuzzily scattered throughout the image. This fuzz is distinctly visible up until the sampling rate of 1024, which it in of itself is a big jump from the sample rate of 64. The fuzz does get a lot less distinct at 16 samples though, showing that's enough samples to obtain an average reasonable close to the actual value, withe that actually being negligibly distant at 1024 samples.



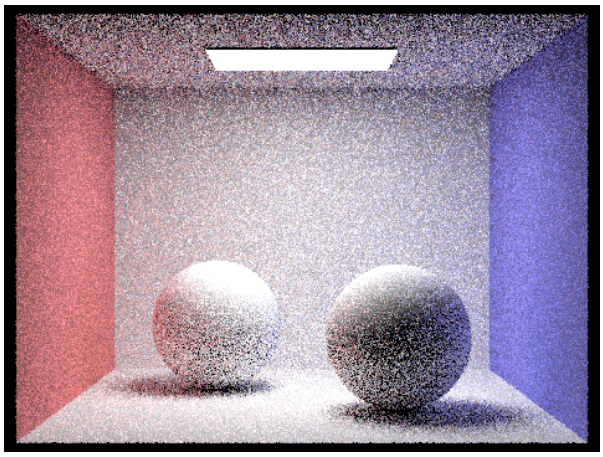
Part 4 Rate 1



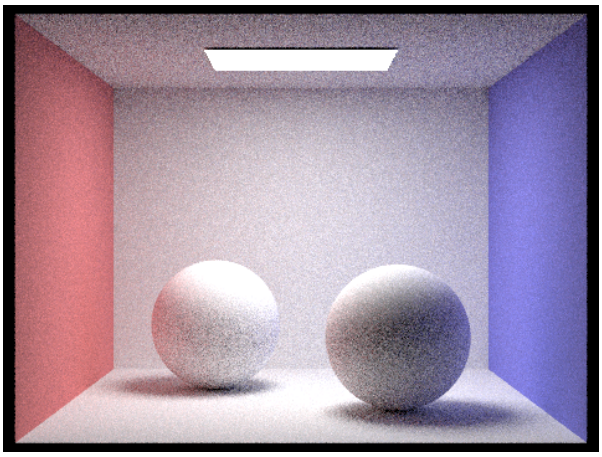
Part 4 Rate 2



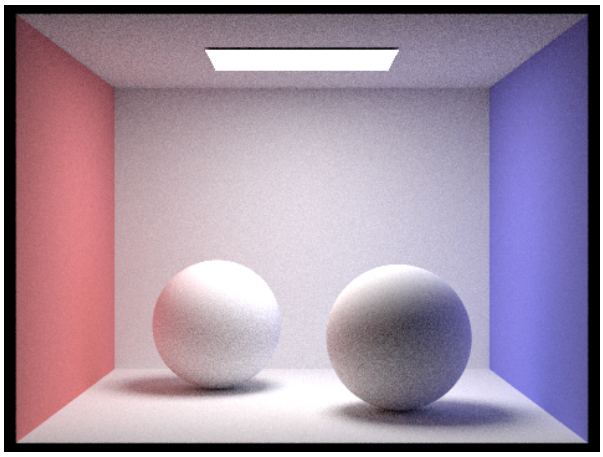
Part 4 Rate 4



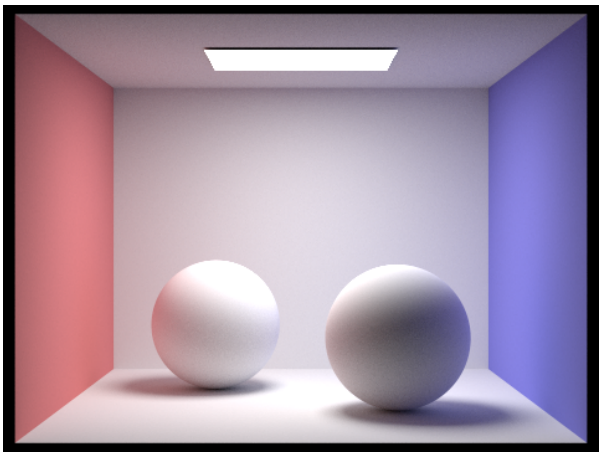
Part 4 Rate 8



Part 4 Rate 16



Part 4 Rate 64



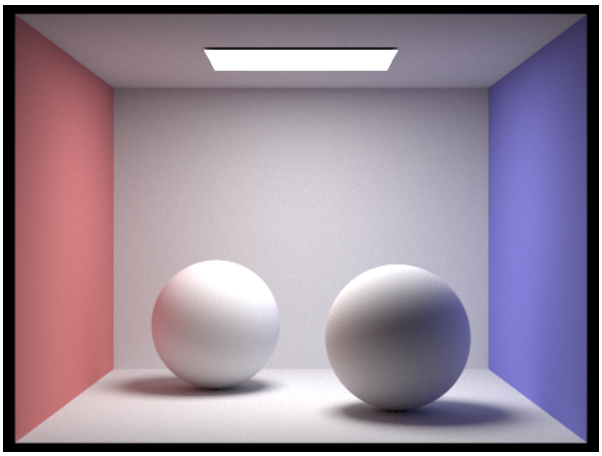
Part 4 Rate 1024

Part 5: Adaptive Sampling

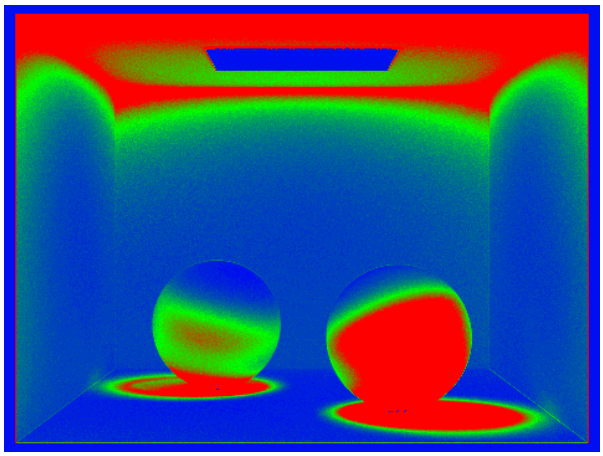
Adaptive Sampling

Adaptive sampling is ensuring the sample rate at different pixels changes to what is needed. Not all points need an insanely high number of pixels to find a good average to stop at, the pathtracing program can instead realize the average value it's obtaining is no longer noticeably changing beyond a reasonable limit and stop taking useless samples. The adaptive sampling we implemented for this project works by first finding a value I that is 1.96 (chosen for 95% confidence) and multiplies by the illuminance standard deviation of the previous samples divided by the number of samples so far. Then, this I value is compared to a selected maximum tolerace value (0.05 for this project) multiplied by the illumincance mean of the previous samples. If I is less than or equal to this value that it is being compared to, then the pixel has likely stopped changing by a noticeable amount for each sample and it is not longer worth it to take more samples, being within I of the mean. To calculate these mean and standard deviation values each time without having to keep track of each separate sample illuminance. values s1 and s2 are obtained where s1 is the sum of the illumance for each sample and s2 is the sum of the square of the illuminance for each sample. The mean can then be obtained by dividing s1 by the number of samples so far and the standard deviation can be obtained by taking 1 divided by the number of samples so far minus one and multiplying that by s2 minus s1 squared over the number of samples so far. In our implementation, we ran this check every 32 samples to avoid slowing the program down by doing it too often.

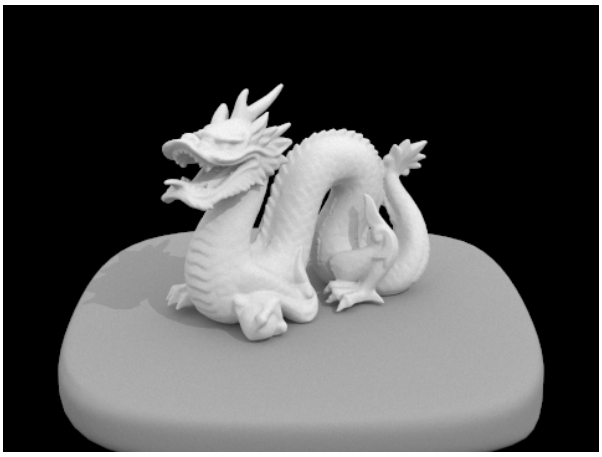
Example Images with their Sample Rates per Pixel



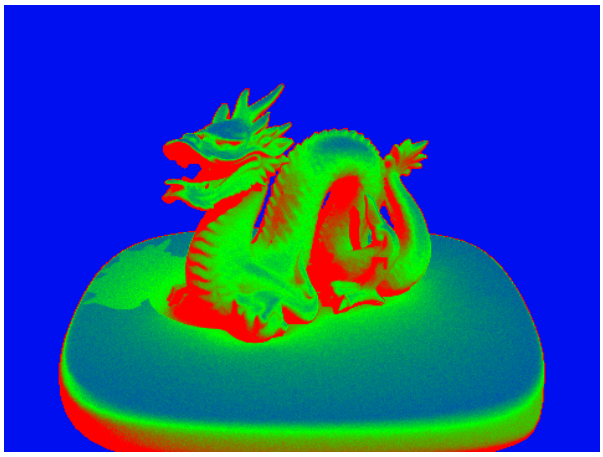
Part 5 CBspheres



Part 5 CBspheres Rate



Part 5 Dragon



Part 5 Dragon Rate

Collaboration with Partner

We collaborated by going over the spec for each part of this assignment together and discussing what approach we should take, before implementing it using zed sharing so that we could both work on the same file at the same time and work on different parts of the planned implementation for each task together. This collaboration overall went smoothly, with some hiccups when writing on the same line sometimes and when debugging sections that required a lot of code, as there was some confusion sometimes when deciphering each other's code. Overall, we learned a lot about how pathtracing works to generate images and the many parts that go into figure out what each pixel should look like while accounting for all the factors evenly and with minimal biasing while also trying to ensure it runs in a short or at least somewhat reasonable amount of time.