CS 184: Computer Graphics and Imaging, Spring 2023

Project 3-1: Path Tracer

Brian Santoso, CS184 Team: #1-beabadoobee-fan

Webpage: https://briansantoso.github.io/project-webpages-sp23-BrianSantoso/proj3/index.html

Overview

In this project I implemented a path tracer renderer including a bounding volume hierarchy (BVH) for ray intersection acceleration, hemisphere and importance direct lighting, global illumination, and adaptive sampling.

In Part 1, I implement primary camera ray generation and primitive (triangle, sphere) intersection.

In Part 2, I construct a BVH with mean centroid position as the splitting point heuristic to accelerate ray-scene intersections.

In Part 3, I implement direct lighting using 2 methods, uniform hemisphere sampling and light importance sampling.

In Part 4, I implement global illumination with Russian Roulette.

In Part 5, I implement adaptive sampling to non-uniformly allocate samples to areas of the image that will need more or less samples to converge to the right answer.

Part 1

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

To generate a camera ray given normalized image coordinates `(x, y)` (`[0, 1] x [0, 1]`), I first compute the position of the bottom left corner of the camera sensor in camera

space, which is equal to `(-half_width, -half_height, -1)` where half_width and half_height are equal to `tan(radians(hFov) / 2)` and `tan(radians(vFov) / 2)` respectively.

I then compute the location of the sensor point in camera space corresponding to normalized image coordinates. This can be done by scaling `(x, y)` by the width and height of the camera sensor and adding them to the bottom left corner of the camera sensor.

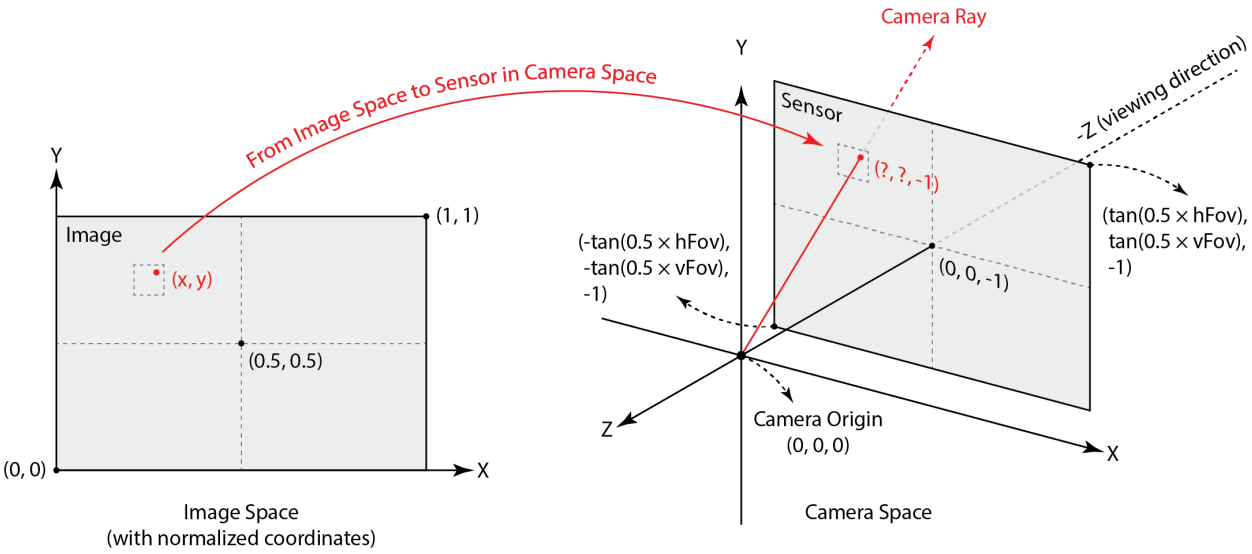


Image from https://cs184.eecs.berkeley.edu/sp23/docs/proj3-1-part-1

Finally I transform the sensor point to world-space by multiplying by the camera-to-world space rotation matrix before translating by the camera position.

Now that I have the camera position and sensor point corresponding to the normalized image coordinates, the camera ray can be easily initialized with its origin being the camera's position and its direction being found by the normalizing vector that starts at camera's position and passes through the sensor point. Finally, I assign lower and upper bounds to the ray's `t` parameter equal to the camera's near clipping and far clipping planes, respectfully.

Once a ray is cast into the scene, it is tested for intersections against the primitive shapes in the scene, which at this point are triangles and spheres. If

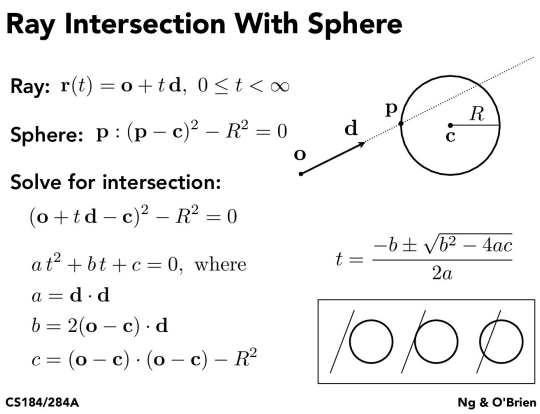
multiple primitives intersect the ray, the intersection used is the one closest to the camera (with the smallest `t` parameter).

An intersection is represented as a struct containing helpful information, including the primitive that caused the intersection, the `t` parameter, the normal of the surface at the point of intersection, and the BSDF of the surface corresponding to the intersection.

It is important to note that an intersection is only considered valid if the `t` parameter falls within the ray's `min_t` and `max_t` fields.

A ray and a given sphere can have up to 2 intersections. We are only interested in the closest valid intersection to the camera.

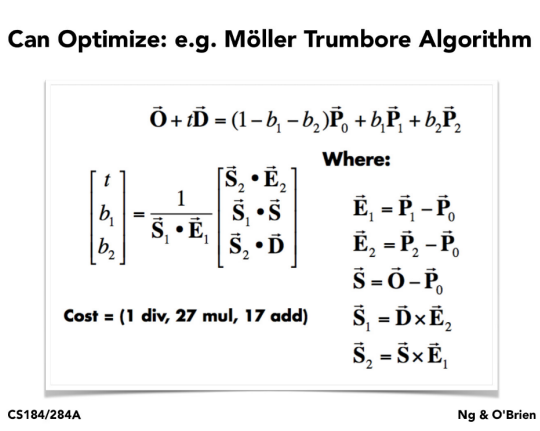
The ray-triangle intersection algorithm is described in the next section.



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

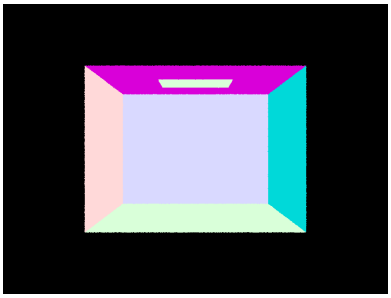
To implement ray-triangle intersection I used the Möller–Trumbore algorithm.

Given a ray and a triangle's vertices, the algorithm computes the parameter t and the barycentric coordinates of a ray-triangle intersection, if there is one. This is done by calculating the normal vector of plane the triangle lies on (recall that any 3 points are coplanar), then finding if the ray intersects the plane, and if so,



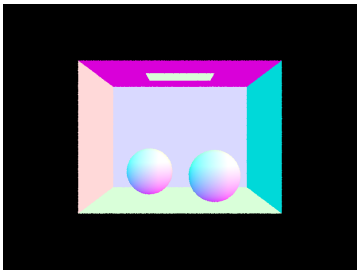
whether the intersection point lies within the 3 edges triangle using the barycentric coordinates of the intersection point.

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



Simple rendering of CBempty.dae with normal shading

`./pathtracer -r 800 600 -f CBempty.png ../dae/sky/CBempty.dae`



Simple rendering of CBsperes.dae with normal shading

`./pathtracer -r 800 600 -f CBspheres.png ../dae/sky/CBspheres.dae`

Simple rendering of cow.dae with normal shading

`./pathtracer -r 800 600 -f cow.png ../dae/meshedit/cow.dae`

Part 2

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

To recursively construct a BVH given a list of primitives, I first compute the bounding box containing all primitives. This corresponds to the bounding box of the current BVH node.

If there are at most `max_leaf_size` primitives, then the current node is a leaf node and I return a newly constructed node containing the given list of primitives.

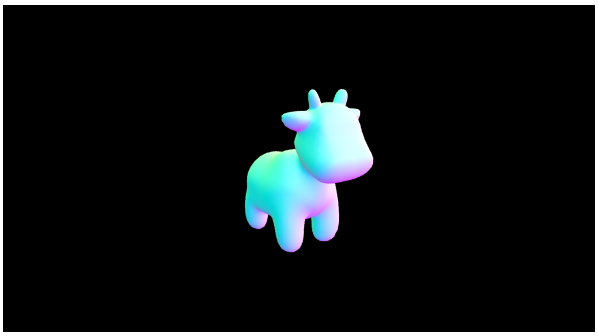
Otherwise. the node is an interior BVH node, in which case I must partition the list of primitives into 2 halves, left and right, based on a splitting point. The heuristic I use to pick the splitting point is the mean centroid position. To do this, I first sort all the objects

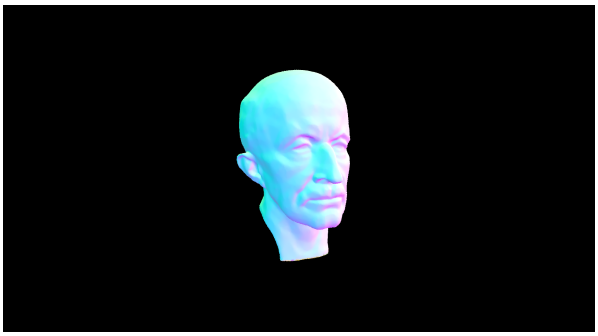
along the longest axis of the current node's bounding box. Then I compute the average position of the given primitives' centroids and partition the objects with a plane perpendicular to the longest axis of the current node's bounding box, containing the average centroid.

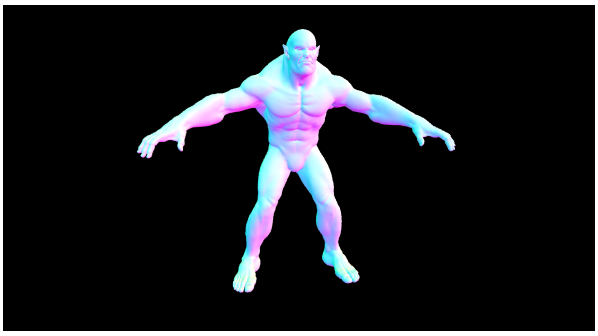
It is possible that after partitioning, the polygons will lie on the same side of the plane (one list is full and one list is empty). This case would result in infinite recursion since the left or right child of the current node would contain the same list of primitives. So in this case, I partition the list of primitives into 2 halves after sorting.

Finally, I recursively compute the current node's left and right children nodes with the newly created left and right partitions, respectfully, and return the newly created node.

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







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

The rendering time for `cow.dae` without BVH acceleration is 40 seconds, while with BVH acceleration shortens it to 0.1157 seconds.

The rendering time for `bunny.dae` without BVH acceleration is 110 seconds, while with BVH acceleration shortens it to 1.3492 seconds.

BVH acceleration sees a larger difference in render times the more complex the scene is, i.e. as the number of primitives increase. Scenes that were unrenderable with the exhaustive implementation are now feasibly renderable with a BVH. Scenes with relatively simple geometries see less benefit from BVH acceleration, and rendering an extremely simple scene such as `cube.dae` even sees a slow down from the exhaustive implementation since the time to construct the BVH exceeds the time saved from using it.

Part 3

Walk through both implementations of the direct lighting function.

The hemisphere and importance sampling differ in the method in which incoming ray directions `w_i` are sampled. We use slightly different Monte Carlo estimators since the sampling function and probability distribution of samples differs.

Hemisphere Sampling

We take a number of samples equal to the light sampling rate * the number of lights in the scene.

We uniformly sample incoming ray directions `w_i` from the unit hemisphere at the surface at the point of intersection.

For each sampled incoming ray direction, we calculate the reflectance using `w_i` and the bsdf of the surface at the point of intersection. Then we calculate the zero-bounce radiance by casting an incoming ray from the hit point with the sampled direction. If the ray intersects a non-emissive surface, then the incoming radiance from this direction will

be 0, manifesting in a shadow. Then we calculate the estimate of outgoing light in the direction of `w_out` as a product of this incoming radiance, the bsdf of the surface given the sampled direction, the cosine of the angle of the sample direction and the surface normal, and normalizing by the the pdf of the sample.

After accumulating the radiance from all samples, we normalize by the number of samples taken.

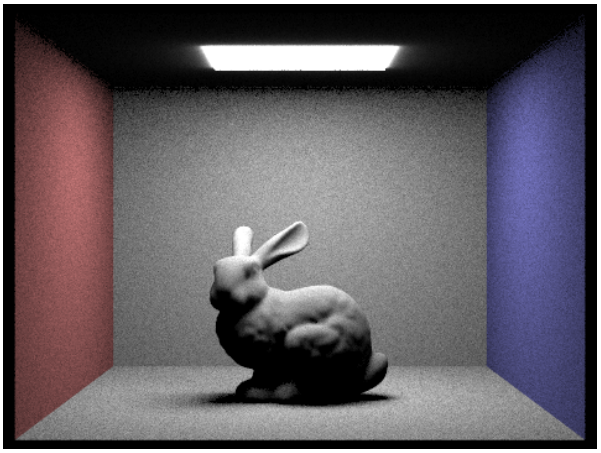
Importance Sampling the Lights

We directly sample each light source in the scene.

If the light is a point light, we only take 1 sample from the light source since each sample will give the same radiance, otherwise if it is an area light we `ns_area_light` samples from the light source.

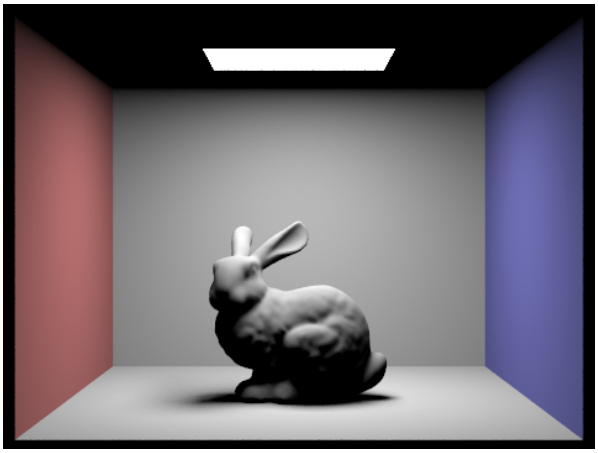
For each light source, and for each sample, we importance sample the light source to get a sample direction and incoming radiance. Then we cast a “shadow” ray starting from the hit point in the direction of the sampled direction. If the ray intersects anything other than the light, then the surface is blocked and the incoming radiance from this direction is 0. Otherwise, we calculate the estimate of outgoing light as a product of the incoming radiance from the light sample, the bsdf of the surface given the sampled direction, the cosine of the angle of the sample direction and the surface normal, and normalizing by the the pdf of the sample. After finishing all samples for a light source, we normalize by the number of sample taken.

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



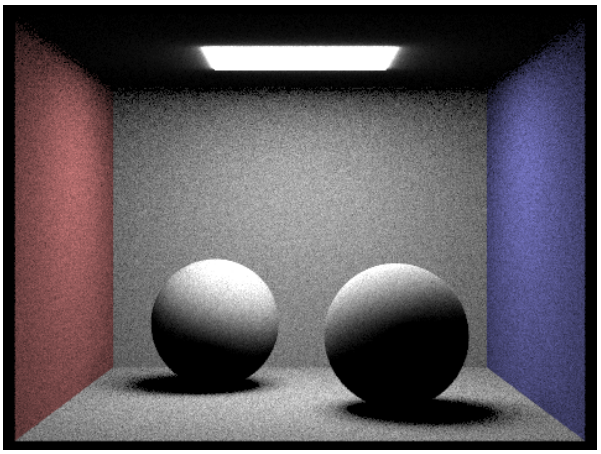
Direct lighting with uniform hemisphere sampling

`./pathtracer -t 8 -s 64 -l 32 -m 6 -H -f CBbunny_H_64_32.png -r 480 360 ../dae/sky/CBbunny.dae`



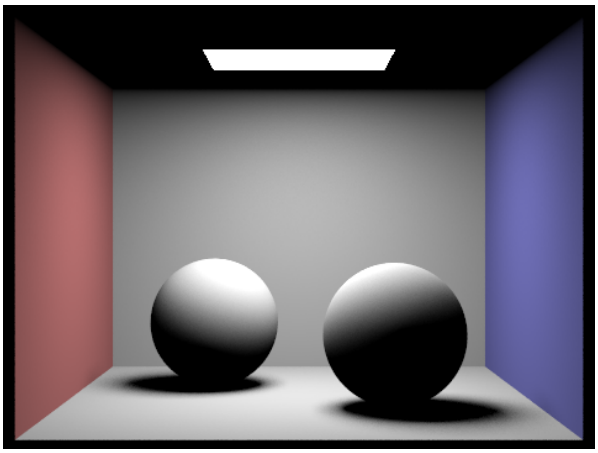
Direct lighting with light importance sampling

`./pathtracer -t 8 -s 64 -l 32 -m 6 -f bunny_64_32.png -r 480 360 ../dae/sky/CBbunny.dae`



Direct lighting with uniform hemisphere sampling

`./pathtracer -t 8 -s 64 -l 16 -m 5 -H -r 480 360 -f spheres.png ../dae/sky/CBspheres_lambertian.dae`



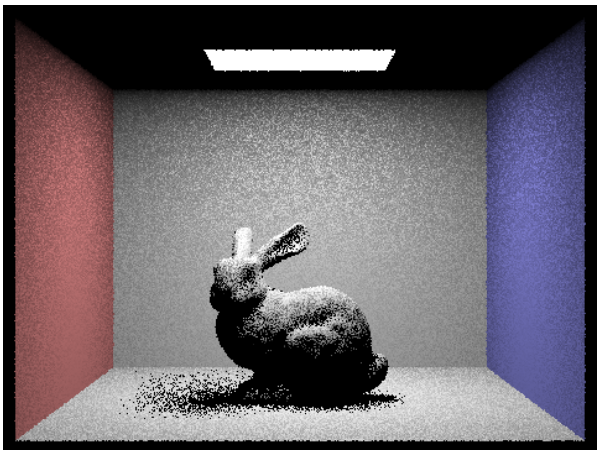
Direct lighting with light importance sampling

`./pathtracer -t 8 -s 64 -l 16 -m 5 -r 480 360 -f spheres.png ../dae/sky/CBspheres_lambertian.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.

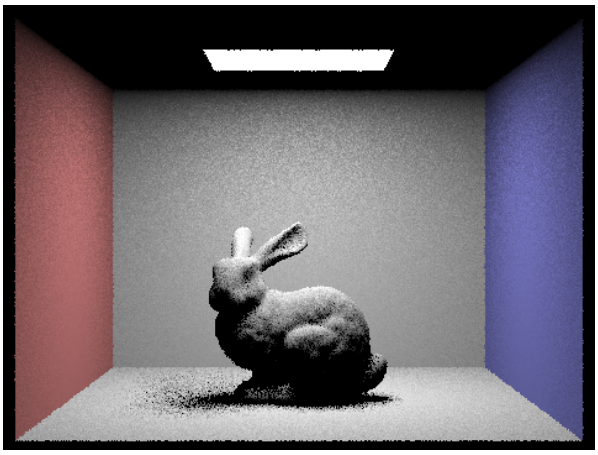
As the number of light rays increase in hemisphere sampling, the noise level of the soft shadows decrease. With a low number of samples, the soft shadows are noisy and the region of the soft shadow appears large, becoming noisier as it moves farther from the

object. But when the light ray samples increase, the soft shadow region converges to the the actual region.



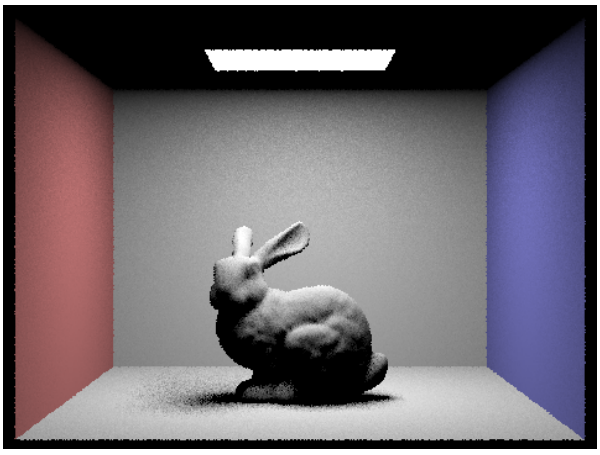
1 light rays, 1 sample per pixel, light sampling

`./pathtracer -t 8 -s 1 -l 1 -m 6 -f bunny_1_64_64.png -r 480 360 ../dae/sky/CBbunny.dae`



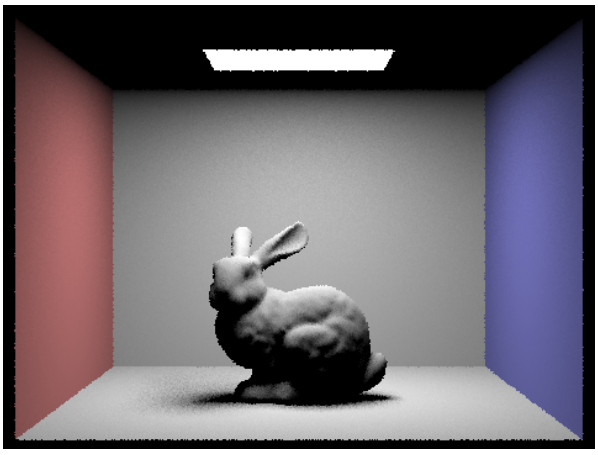
4 light rays, 1 sample per pixel, light sampling

`./pathtracer -t 8 -s 1 -l 4 -m 6 -f bunny_1_64_64.png -r 480 360 ../dae/sky/CBbunny.dae`



16 light rays, 1 sample per pixel, light sampling

`./pathtracer -t 8 -s 1 -l 16 -m 6 -f bunny_1_64_64.png -r 480 360 ../dae/sky/CBbunny.dae`



64 light rays, 1 sample per pixel, light sampling

`./pathtracer -t 8 -s 1 -l 64 -m 6 -f bunny_1_64_64.png -r 480 360 ../dae/sky/CBbunny.dae`

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

Importance sampling the light reduces the total noise per sample.

Uniform hemisphere lighting will not accumulate radiance from point light sources since the probability that a ray sampled uniformly from the unit hemisphere hits a point light source is essentially 0.

As the number of light rays increase in hemisphere sampling, the noise level of the soft shadows decrease. With a low number of samples, the soft shadows are noisy and the region of the soft shadow appears large, becoming noisier as it moves farther from the object. But when the light ray samples increase, the soft shadow region converges to the the actual region.

Light sampling converges quicker to the answer than uniform hemisphere sampling.

Part 4

Walk through your implementation of the indirect lighting function.

We partition the rendering into 2 parts: calculating the zero-bounce radiance and calculating the at-least-one bounce radiance.

The zero-bounce radiance was implemented in the previous parts and is calculating the light that results from no bounces of light, essentially just rendering only the emissive surfaces.

The indirect lighting function is implemented in at-least-one bounce radiance. It is a recursive process.

At each ray depth, when a ray intersects a surface we calculate the one-bounce radiance arriving at the intersection—in other words, the radiance resulting directly from the lights in the scene.

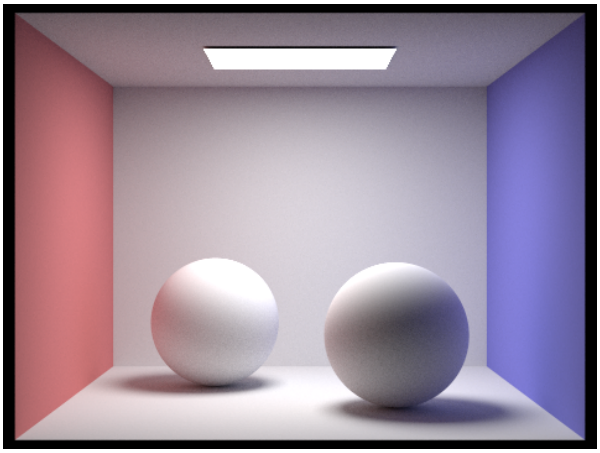
Then, we begin accumulating the additional incoming radiance at the point that results from the indirect lighting of the scene (incoming light that requires more than one bounce to arrive at this point). We estimate the indirect light by the sampling incoming rays via a combination of Monte Carlo estimation and Russian Roulette. For each sample, we flip a coin with a probability `continuation_prob`—if the coin lands head we continue with the sample, otherwise we discard the current sample and move on to the next (Russian Roulette).

If the coin lands head, we generate an incoming ray and direction by importance sampling the bsdf of the current surface of the intersection. We estimate the incoming

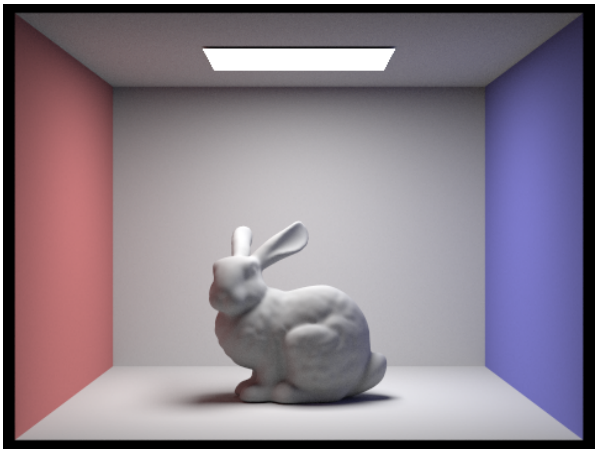
radiance from this sample ray by recursively calling the indirect lighting function (at-least-one bounce radiance) with the incoming ray. Then we calculate the outgoing light from this ray direction as a product of this incoming radiance, the bsdf of the surface given the sampled direction, cosine of the angle of the sample direction and the surface normal, and normalizing by both the the pdf of the sample and `continuation_prob`.

If the number of ray bounces reaches the max ray depth or the coin lands tails, then we stop recursing.

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



`./pathtracer -t 8 -s 1024 -l 16 -m 100 -r 480 360 -f spheres_100.png ../dae/sky/CBspheres_lambertian.dae`

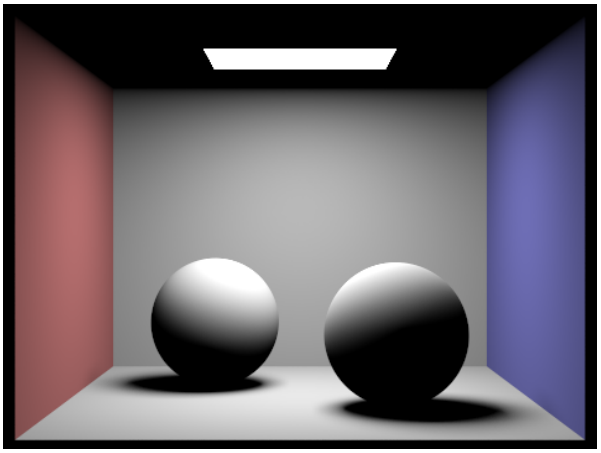


`./pathtracer -t 8 -s 1024 -l 16 -m 2 -r 480 360 -f bunny_2.png ../dae/sky/CBbunny.dae`

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.)

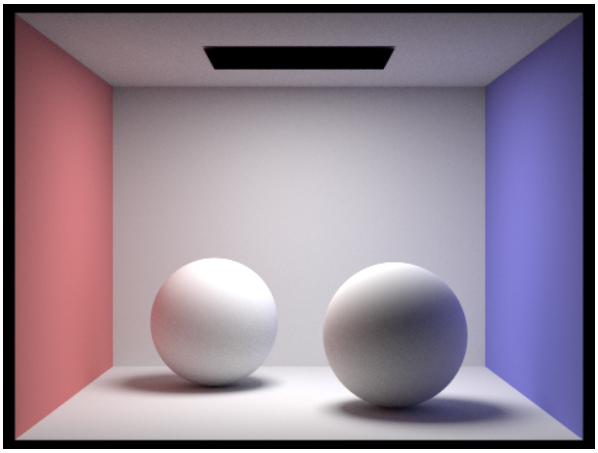
The direct illumination only has no indirect lighting and we see no color bleeding. The light is visible.

The indirect lighting has color bleeding and smoother shadows, but the light itself is black.



Only direct illumination (zero bounce and one bounce)

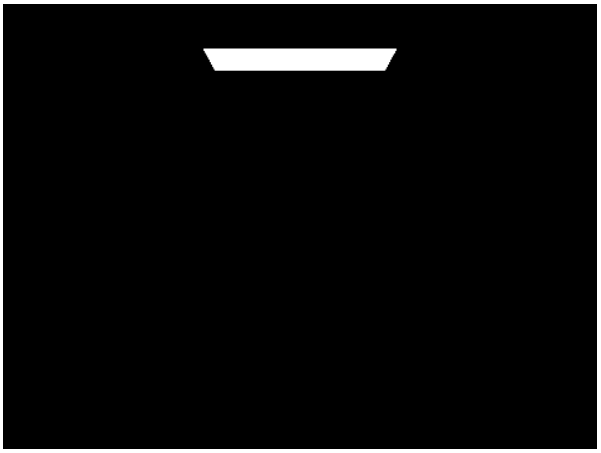
`./pathtracer -t 8 -s 1024 -l 16 -m 0 -r 480 360 -f sphere_directonly.png ../dae/sky/CBspheres_lambertian.dae`



Only indirect illumination (at least once bounce - one bounce - zero bounce) with max ray depth of 6

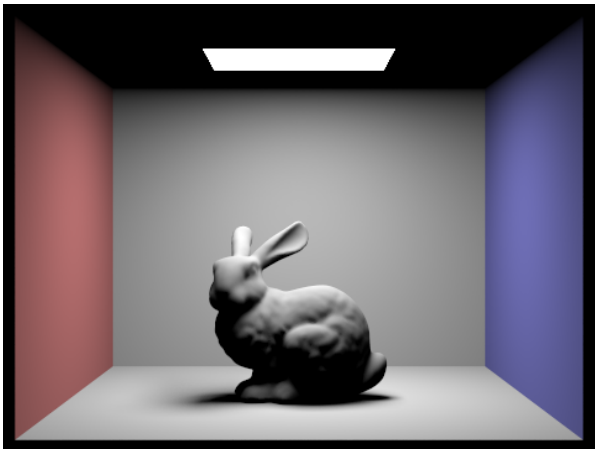
For *CBbunny.dae*, compare rendered views with `max_ray_depth` set to 0, 1, 2, 3, and 100 (the `m` flag). Use 1024 samples per pixel.

With a max ray depth of 0, we only get the zero-bounce lighting. With a max ray depth of 1, we get zero-bounce and direct lighting. With a max ray depth ≥ 2, we get global illumination. As the max ray depth increases after this, we can a more accurate answer, but the benefits of increasing the max ray depth taper off.



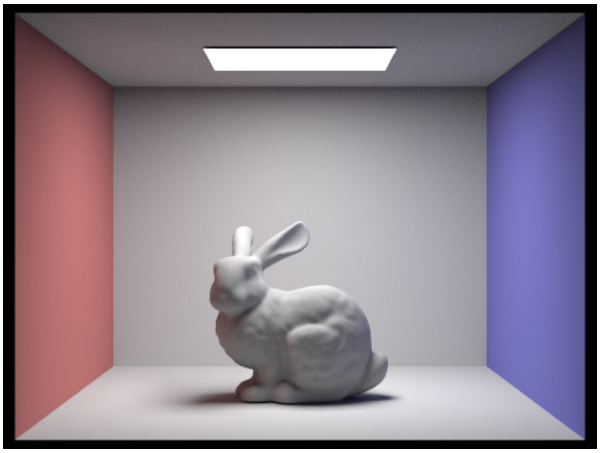
Max ray depth of 0, 1024 samples per pixel

`./pathtracer -t 8 -s 1024 -l 16 -m 0 -r 480 360 -f bunny_0.png ../dae/sky/CBbunny.dae`



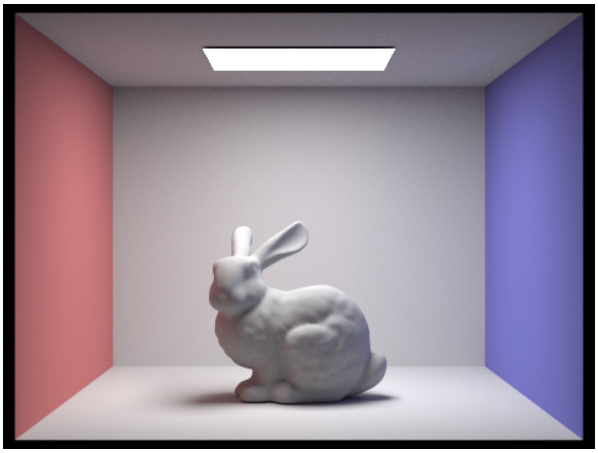
Max ray depth of 1, 1024 samples per pixel

`./pathtracer -t 8 -s 1024 -l 16 -m 1 -r 480 360 -f bunny_1.png ../dae/sky/CBbunny.dae`



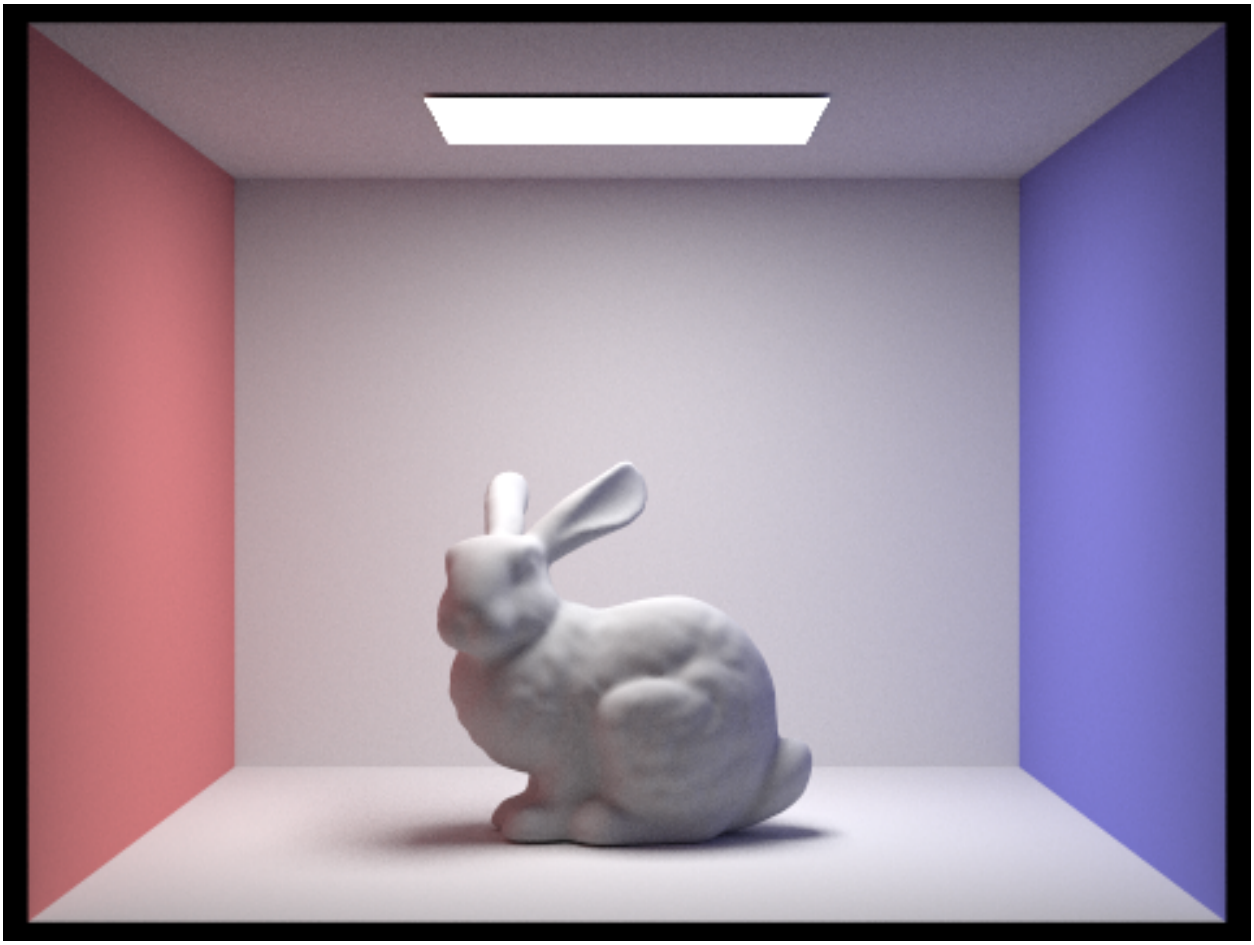
Max ray depth of 2, 1024 samples per pixel

`./pathtracer -t 8 -s 1024 -l 16 -m 2 -r 480 360 -f bunny_2.png ../dae/sky/CBbunny.dae`



Max ray depth of 3, 1024 samples per pixel

`./pathtracer -t 8 -s 1024 -l 4 -m 3 -r 480 360 -f bunny_3.png ../dae/sky/CBbunny.dae`

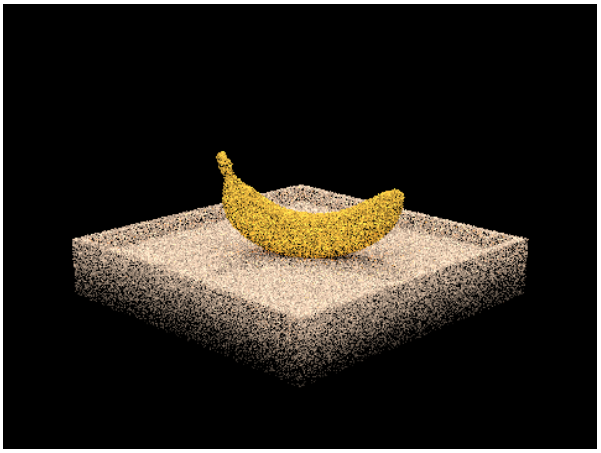


Max ray depth of 100, 1024 samples per pixel

```
./pathtracer -t 8 -s 1024 -l 1 -m 100 -r 480 360 -f bunny_100.png ../dae/sky/CBbunny.dae
```

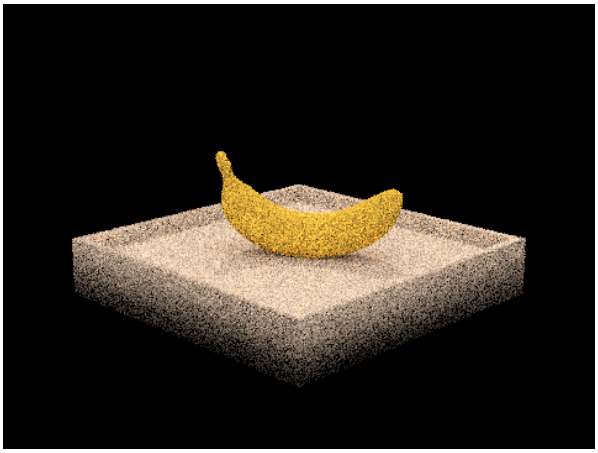
Pick one scene and compare rendered views with various sample-per-pixel rates, including at least 1, 2, 4, 8, 16, 64, and 1024. Use 4 light rays.

As the sample-per-pixel rates increase, the noise decreases and the closer we get to the answer.



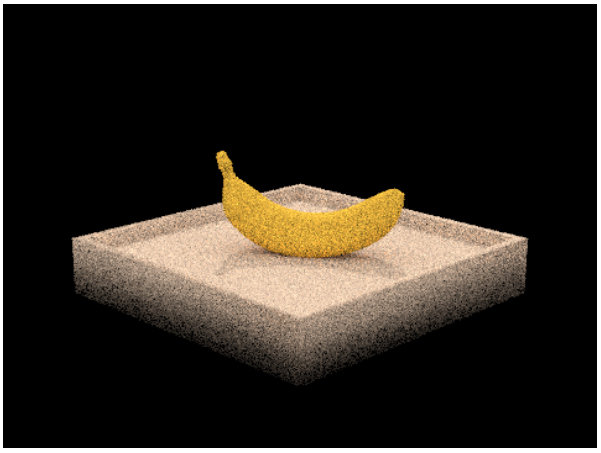
1 sample per pixel

`./pathtracer -t 8 -s 1 -l 4 -m 6 -r 480 360 -f banana_1.png ../dae/keenan/banana.dae`



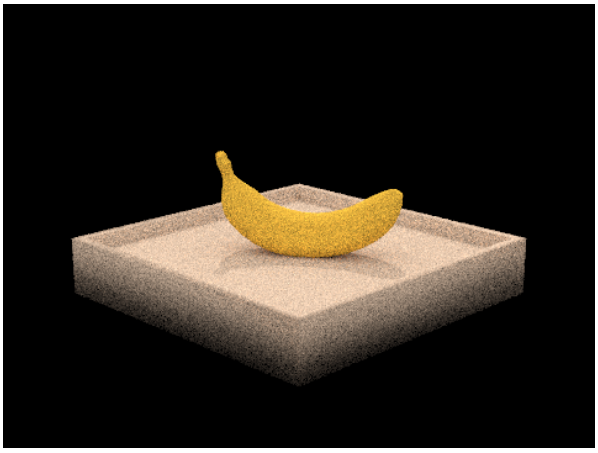
2 samples per pixel

`./pathtracer -t 8 -s 2 -l 4 -m 6 -r 480 360 -f banana_2.png ../dae/keenan/banana.dae`



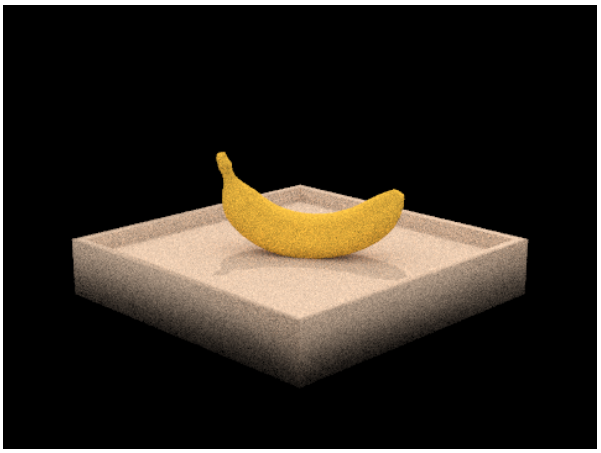
4 samples per pixel

`./pathtracer -t 8 -s 4 -l 4 -m 6 -r 480 360 -f banana_4.png ../dae/keenan/banana.dae`



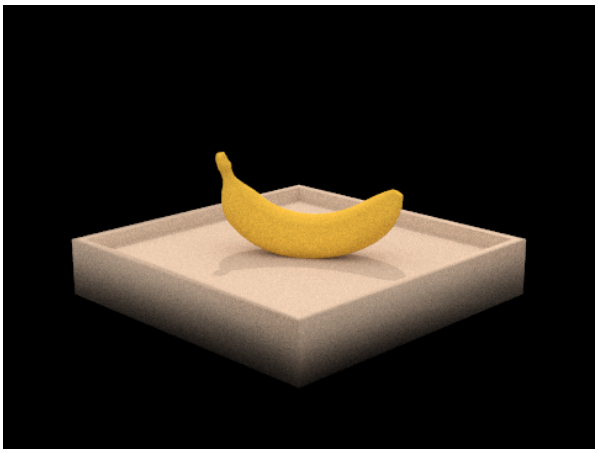
8 samples per pixel

`./pathtracer -t 8 -s 8 -l 4 -m 6 -r 480 360 -f banana_8.png ../dae/keenan/banana.dae`



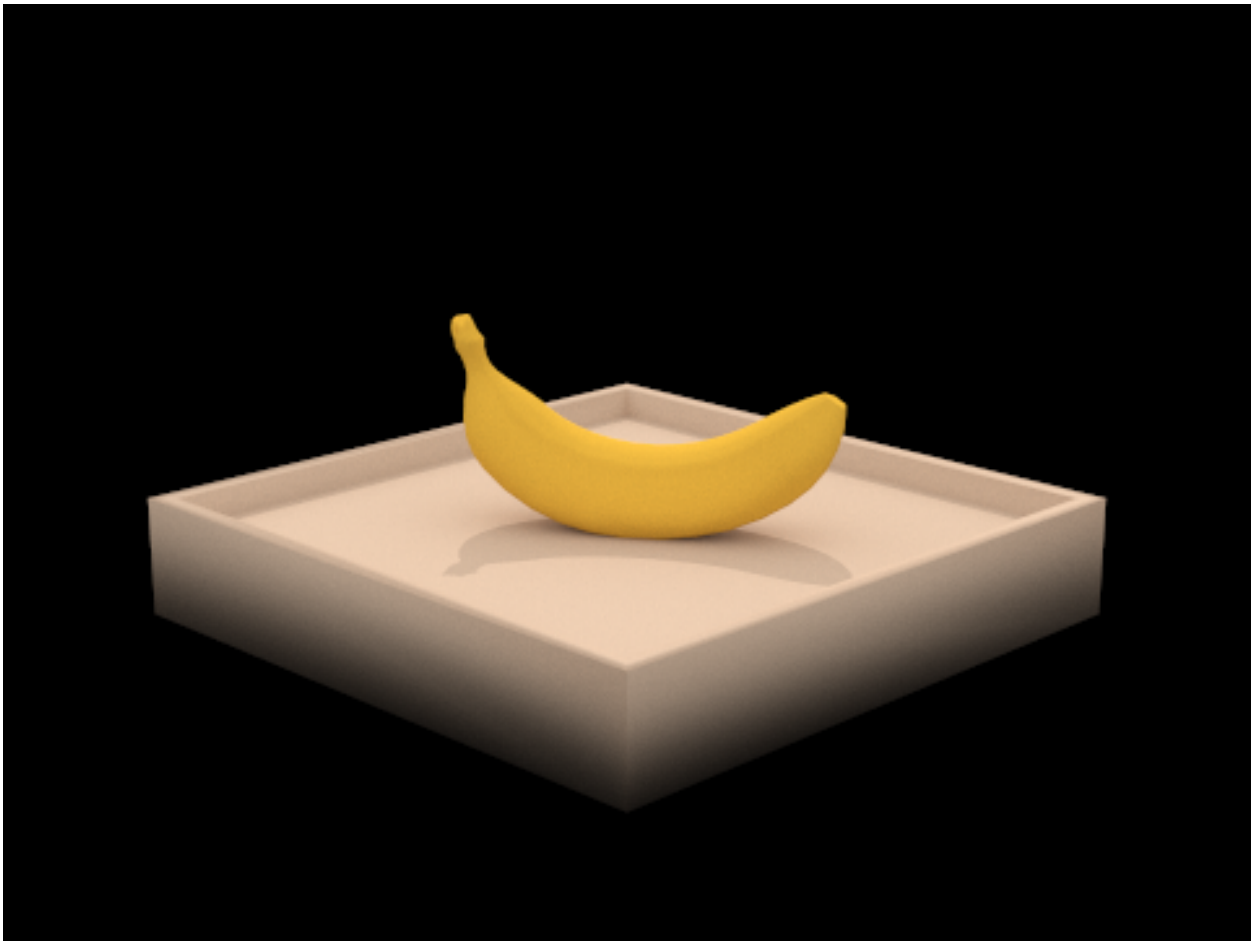
16 samples per pixel

`./pathtracer -t 8 -s 16 -l 4 -m 6 -r 480 360 -f banana_16.png ../dae/keenan/banana.dae`



64 samples per pixel

`./pathtracer -t 8 -s 64 -l 4 -m 6 -r 480 360 -f banana_64.png ../dae/keenan/banana.dae`



1024 samples per pixel

`./pathtracer -t 8 -s 1024 -l 4 -m 6 -r 480 360 -f banana_1024.png ../dae/keenan/banana.dae`

Part 5

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

Adaptive sampling is a method to non-uniformly allocate samples to areas of the image that will need more or less samples to converge to the right answer. By redistributing the samples with adaptive sampling, we can reduce the total noise for a given number of samples.

For each pixel, we can calculate a statistical measure of the its convergence `I` by keeping track of the number of samples taken so far and calculating the mean and standard deviation of the illuminance of the pixel so far.

$$I = 1.96 \cdot \sigma / \sqrt{n}$$

When `I` is small, we have more confidence that the pixel has converged. We can stop taking samples when we are confident enough that the pixel is converged with the following condition:

$$I \leq maxTolerance \cdot \mu$$

In my implementation, I keep track of 2 variables `s1`, `s2`, corresponding to the sum of the illuminance so far and the sum of the squared illuminance so far. Every `samplesPerBatch`, samples I calculate the mean, variance, and standard deviation using `s1`, `s2` and check the above termination criteria.

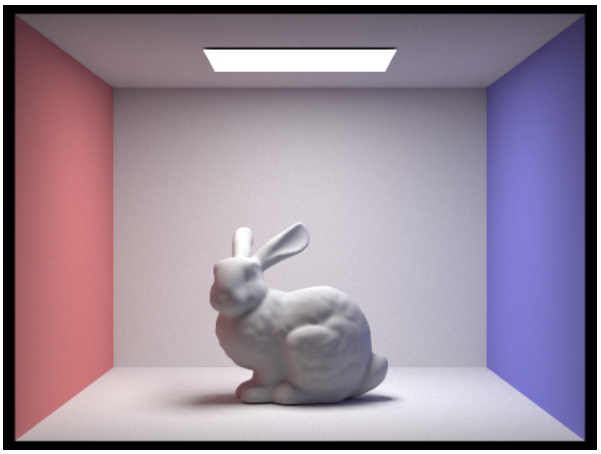
$$s_1 = \sum_{k=1}^{n} x_k,$$
$$s_2 = \sum_{k=1}^{n} x_k^2,$$

$$\mu = \frac{s_1}{n}$$

$$\sigma^2 = \frac{1}{n-1} \cdot \left(s_2 - \frac{s_1^2}{n} \right)$$

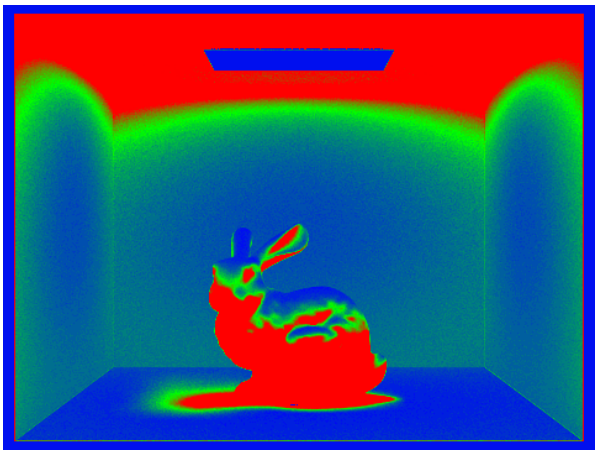
Pick two scenes and render them with at least 2048 samples per pixel. Show a good sampling rate image with clearly visible differences in sampling rate over various regions and pixels. Include both your sample rate image, which shows your how your adaptive sampling changes depending on which part of the image

you are rendering, and your noise-free rendered result. Use 1 sample per light and at least 5 for max ray depth.

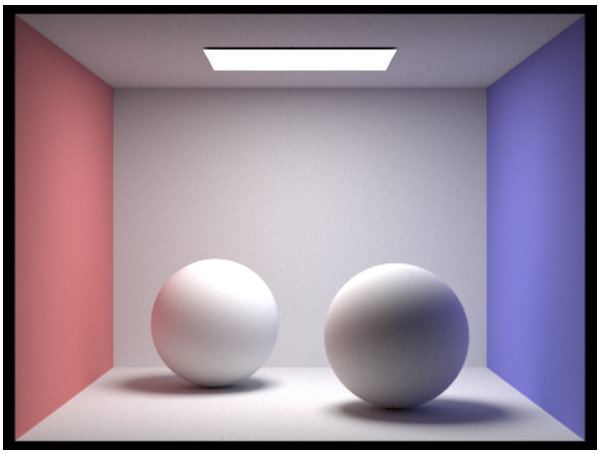


Bunny with adaptive sampling. 2048 samples per pixel, 1 sample per light, 5 max ray depth

```
./pathtracer -t 8 -s 2048 -a 64 0.05 -l 1 -m 5 -r 480 360 -f bunny.png ../dae/sky/CBbunny.dae
```

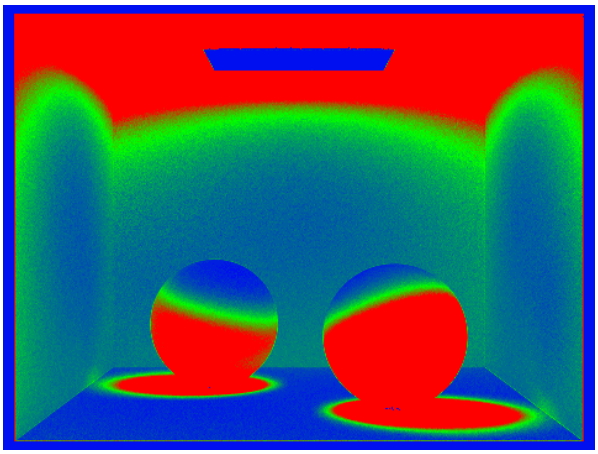


Sample rates per pixel for Bunny render with adaptive sampling



Spheres with adaptive sampling. 2048 samples per pixel, 1 sample per light, 5 max ray depth

```
./pathtracer -t 8 -s 2048 -a 64 0.05 -l 1 -m 5 -r 480 360 -f spheres.png ../dae/sky/CBspheres_lambertian.dae
```



Sample rates per pixel for Spheres render with adaptive sampling