Assignment 3: PathTracer

Eden Lin

Part 1: Ray Generation and Intersection

Our goal here is to cast some number of rays through a random position in the pixel box. We want to cast *ns_aa* rays, so we repeat the following for *ns_aa* times.

Before we start, we need to initialize a Monte Carlo estimator, *mc*, this is a `Spectrum` value. To start, we get the random position first. I used *gridSampler* to get a random sample in a unit square, then add it to the given position. This new position is then normalized by dividing the *x* and *y* coordinate by *sampleBuffer*'s width and height, respectively. We pass this position in image space to `Camera::generate_ray(...)` to generate a ray. We then use this ray to call `PathTracer::est_radiance_global_illumination(...)` to estimate its scene radiance. The value returned is a `Spectrum` value, we add it to the Monte Carlo estimator.

After this process has been repeated for *ns_aa* times, we normalized the Monte Carlo estimator by dividing it by *ns_aa* and update the corresponding pixel by calling `update_pixel(...)`.

Now let's rewind and talk about how is a ray generated in `Camera::generate_ray(...)`. First, we know that a ray is consisted of an origin and a direction. The origin is just the origin in camera space, which is *(0, 0, 0)*. To get the direction, we need to convert the pixel position from image space to camera space. How? We know the boundary coordinates of the camera space bounding box. We also know that the coordinates passed into this function is normalized to be between 0 and 1, this coordinate tells the position in image space relative to its width and height. We can use this normalized image coordinates to scale the width and height of the camera space bounding box, add to the minimum coordinates of the camera space bounding box to get the corresponding camera space coordinates.

After we have got the correct camera apace coordinates, we proceed to calculating the direction vector from the camera to that position. Usually, to get a direction vector we need to subtract one vector from another, however, here we always subtract the origin, which is *(0, 0, 0)*, so the value of the vector we subtract the origin from will not change, therefore, the direction vector is just the camera space coordinates we calculated.

Lastly, we set the ray's *min_t* and *max_t* to *nclip* and *fclip*. After we finished this last step, the freshly made ray is now ready to return.

Next, we move on to triangle intersection, the method I am going to use here is the **Möller-Trumbore algorithm**. Before we get into the implementation, let's first derive this formula to see why and how it works.

As we have alreay known, a ray is defined by an origin and a direction, it is expressed in equation as

$$r(t) = O + tD,$$

Note that *t* here must be a non-negative number.

A point within a triangle $\boldsymbol{P_0P_1P_2}$ can be represented by interpolation as

$$P = \alpha P_0 + \beta P_1 + \gamma P_2,$$

where $\alpha + \beta + \gamma = 1$.Given the relationship between α, β, and γ, we can rewrite $\boldsymbol{P}$ as

$$P = (1 - b_1 - b_2)P_0 + b_1P_1 + b_2P_2.$$

So, the question is, how do we solve for the intersection given these equations. That's easy, we just set them equal to each other:

$$O + tD = (1 - b_1 - b_2)P_0 + b_1P_1 + b_2P_2,$$

moving the components around, we write the equation again just in different order:

$$O - P_0 = -tD + b_1(P_1 - P_0) + b_2(P_2 - P_0).$$

We intentionally put all the coefficients on one side so we can write this in matrix format as

$$\begin{bmatrix} -D & P_1 - P_0 & P2 - P_0 \end{bmatrix} \begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = O - P_0.$$

This format is desirable because we can then apply the **Cramer's rule** and solve for the coefficient matrix. In short, after some rearragement and basic calculating, the coefficients can be found by the following matrix equation:

$$\begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = \frac{1}{S_1 \cdot E_1} \begin{bmatrix} S_2 \cdot E_2 \\ S_1 \cdot S \\ S_2 \cdot D \end{bmatrix}$$

where $E_1 = P_1 - P_0$, $E_2 = P_2 - P_0$, $S = O - P_0$, $S_1 = D \times E_2$, $S_2 = S \times E_1$.

This concludes the derivation of the Möller-Trumbore algorithm. Let's now move on to explanation of the sphere intersetion with a ray.

Ray intersection with sphere is actually simpler than triangle to derive. First of all, we know that a point $\boldsymbol{p}$ is only on the sphere if $(\boldsymbol{p} - \boldsymbol{c})^2 - R^2 = 0$ where $\boldsymbol{c}$ is the center and R is the radius of the sphere. Since we want to when does $r(t)$ will be on the sphere, we simply plug it in to the equation for $\boldsymbol{p}$.

$$(o + td - c)^2 - R^2 = 0$$

$$at^2 + bt + c = 0,$$

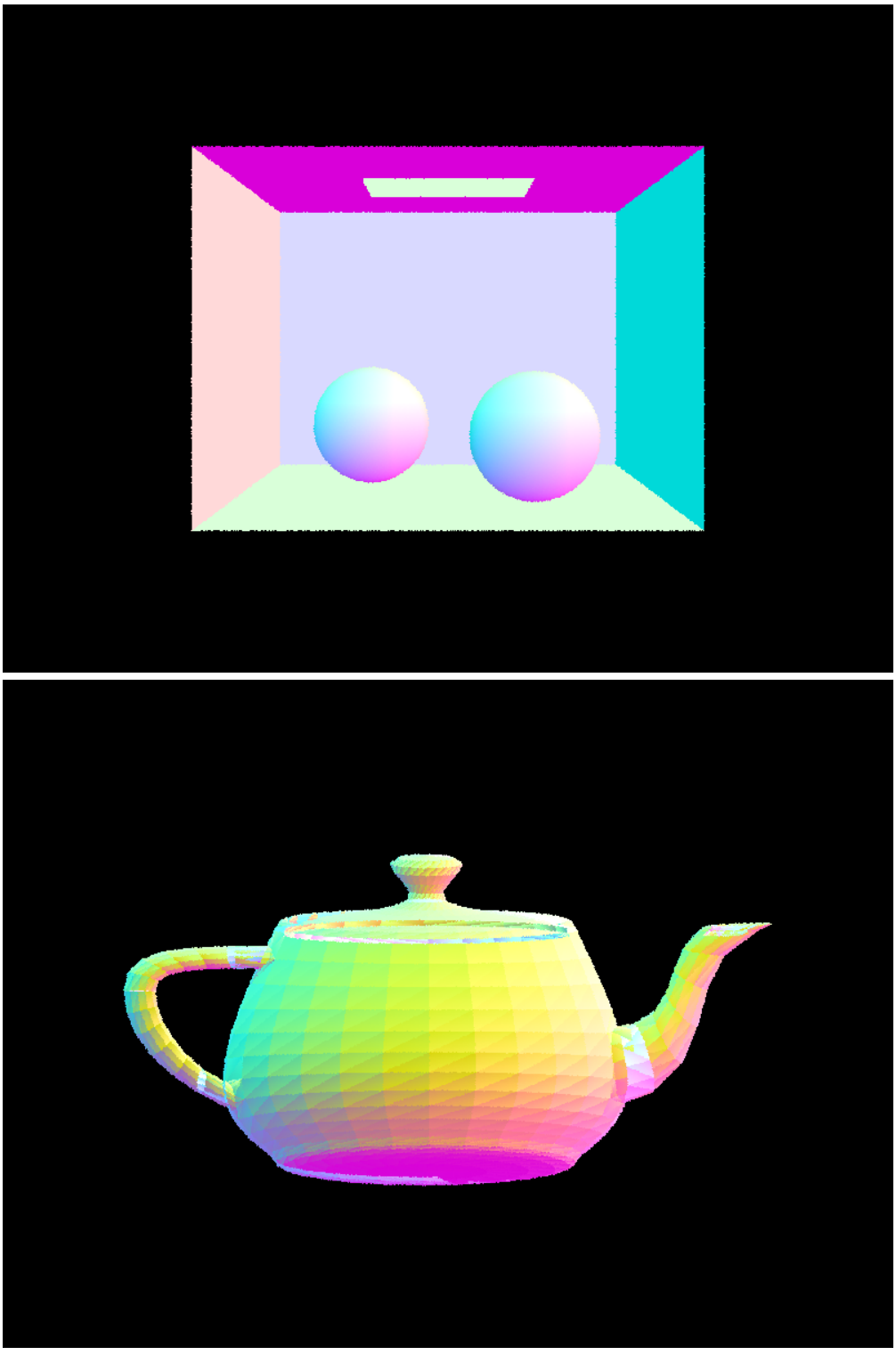
where $a = \boldsymbol{d} \cdot \boldsymbol{d}$, $b = 2(\boldsymbol{o} - \boldsymbol{c}) \cdot \boldsymbol{d}$, $c = (\boldsymbol{o} - \boldsymbol{c}) \cdot (\boldsymbol{o} - \boldsymbol{c}) - R^2$. With these values, we can use the quadratic formula to solve for t:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Note that the solution here can have 0, 1, or 2 roots, what does each of them mean in terms of intersection? When there is 0 root, it means the ray did not hit the sphere at all; when there is 1 root, the ray hits the sphere at a tangent point; when there are 2 roots, the ray passes through the sphere thus there are two intersections, one going in and one going out.

That wraps up the explanation of the intersection algorithm I am going to use. The implementation is very simple in the way that it follows strictly from the algorithm, all I need to do is just some arithmatic calculation. In the end, everything should work as expected as long as correct condition checking is implemented. That is, for triangle intersection, t needs to be in the range of the ray's *min_t* and *max_t*. b_1 and b_2 need to be a value between 0 and 1,

and the relationship between b_1 and b_2 needs to be guaranteed. If any of these conditions are not satisfied, we simply return `false`. If they are, then we set the member variables of the `Intersection` object passed in to their corresponding values. For sphere intersection, we will return `false` if there is no solution to the quadratic formula, or none of the roots is within the valid range.



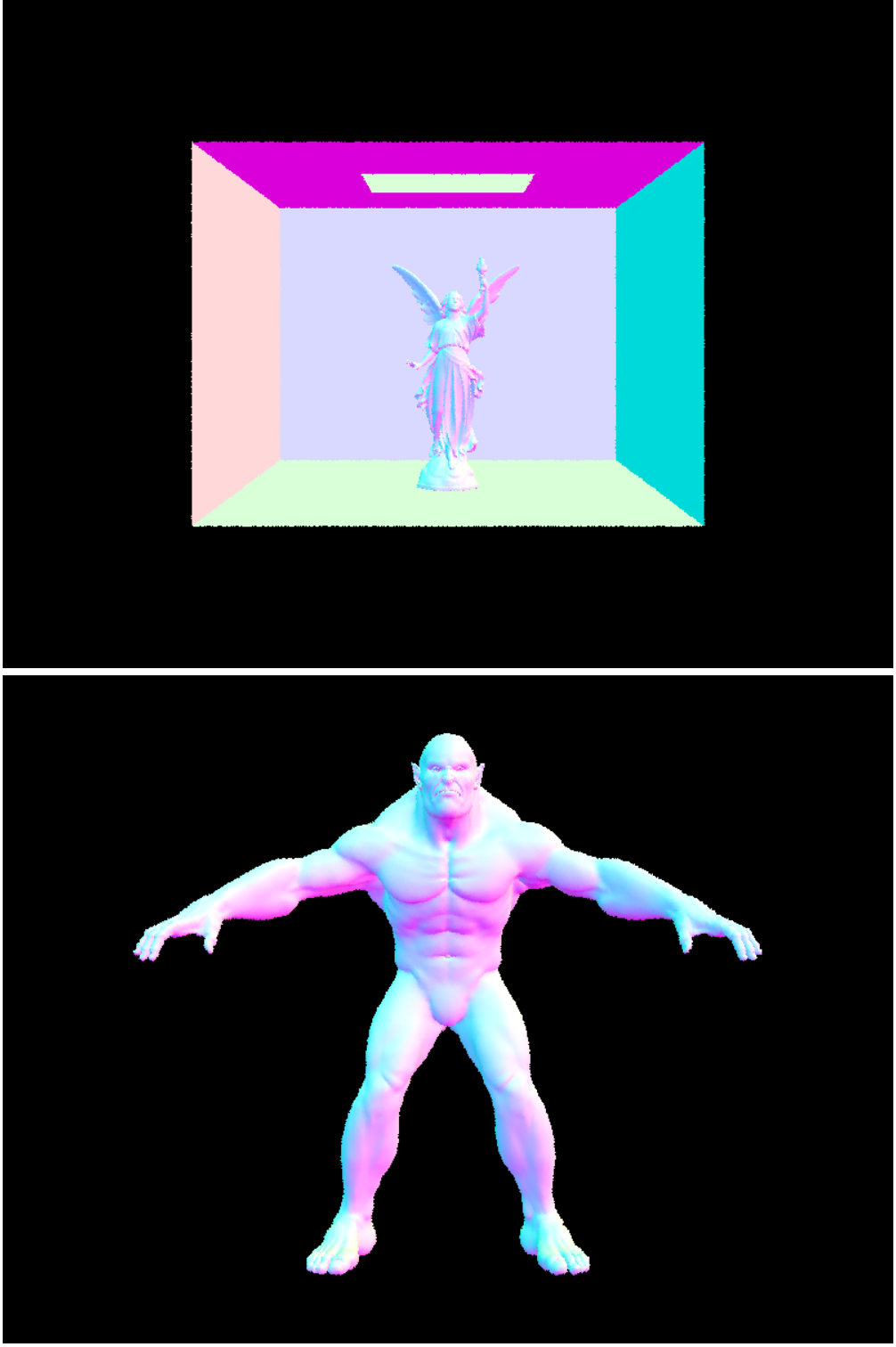
Images with normal shading.

Part 2: Bounding Volume Hierarchy

When constructing a `BVHNode`, we first collect all bounding boxes from the primitives into a `BBox` object using the iterators that were passed in. We then check if the number of primitives is at most *max_leaf_size*, if it is, we make a new `BVHNode` pointer with that `BBox` object, set the node's *start* iterator and *end* iterator to the ones that were passed in, and return the node. If the number of primitives is greater than *max_leaf_size*, we have to decide on a split point,

make two vectors of primitive pointers, and push the primitives on each side of the split point into the left and right child. One thing to note here is that since we need to access all the nodes after we finished constructing the BVH, to make sure the iterators do not go out of scope after the function returns, we need to allocate memory for them on heap by using the `new` keyword. Then, we recursively call `construct_bvh(...)` to build the children. These steps should be straight forward enough, the main problem is, how do we decide the split point?

First, let's make it clear that we are splitting the bounding box along the greatest axis. This information is enclosed in the *extent* of the `BBox` object. At first, I wanted to make it simple, so I used the midpoint on the axis. This heuristic performs okay in the beginning, but later I had encountered an issue with it. Towards the end of the splitting process, there may be a small number of primitives left that is bigger than the *max_leaf_size*, however, they all lie on one side of the midpoint, so one of the children of the node has size zero. To fix this problem, I made it so that it takes the first primitive pointed to by the *start* iterator as an arbitrary split point. This works at first when I tried to render simpler images like *CBspheres.dae*, but it failed when I try to render more complex images like *maxplanck.dae* and *CBlucy.dae*. I print to see the size of the children when splitting, what I had found is that one of the children always has size 1 after some split. At this point, I realized that I must use a more robust splitting point method. So, I changed the heuristic to using the midpoint of the minimum and maximum primitive coordinate on the biggest axis. This means I have to use a for loop to find the biggest and smallest primitive coordinate along an axis, but the good news is, after I had made this change, I never ran into splitting point problem again.



Images render with BVH accerleration.

Let's compare between rendering without BVH and rendering with BVH. See the table below.

Image	Number of Primitives	Rendering Time: without BVH/with BVH
cow.dae	5,856	39.6446s/0.1461s
maxplanck.dae	50,801	830.6946s/0.1904s
CBlucy.dae	133,796	2091.2014s/0.1568s

It is not the case for *cow.dae*, but for *maxplanck.dae* and *CBlucy.dae*, when divide the time in second by the number of primitives, we get a number of about 0.016. This linear time

complexity makes sense since every time we are traversing all primitives. However, with the BVH, we can filter out about hald of the primitives in the node as we traverse down the structure, resulting in a *O(logn)* time complexity. This is a huge improvement! As we can see from the table, the time to render *CBlucy.dae*, which has 133,796 primitives, is almost the same as to render *cow.dae*, which only has 5,856 primitives. This may also imply that the more primitives in an image, the more useful is BVH. This should make sense if you think about it. When there are only a few primitives, filtering out half may not make very notable improvement, but if there are tens of thousands of primitives, the time we saved by leaving out half of them would be huge.

Part 3: Direct Illumination

When dealing with Lambertian material, the light reflected on the surface is always constant. Therefore, the function `DiffuseBSDF::f(...)` always return a constant, which is *reflectance*/π. `DiffuseBSDF::sample_f(...)` does similar thing, but as its name suggested, it also sample an incoming direction *wi*, assign it as well as its corresponding *pdf* to the pointers passed in. `PathTracer::zero_bounce_radiance(...)` is very simple to implement, since there is no bounce, the radiance returned is just the light emitted from the intersection's surface area.

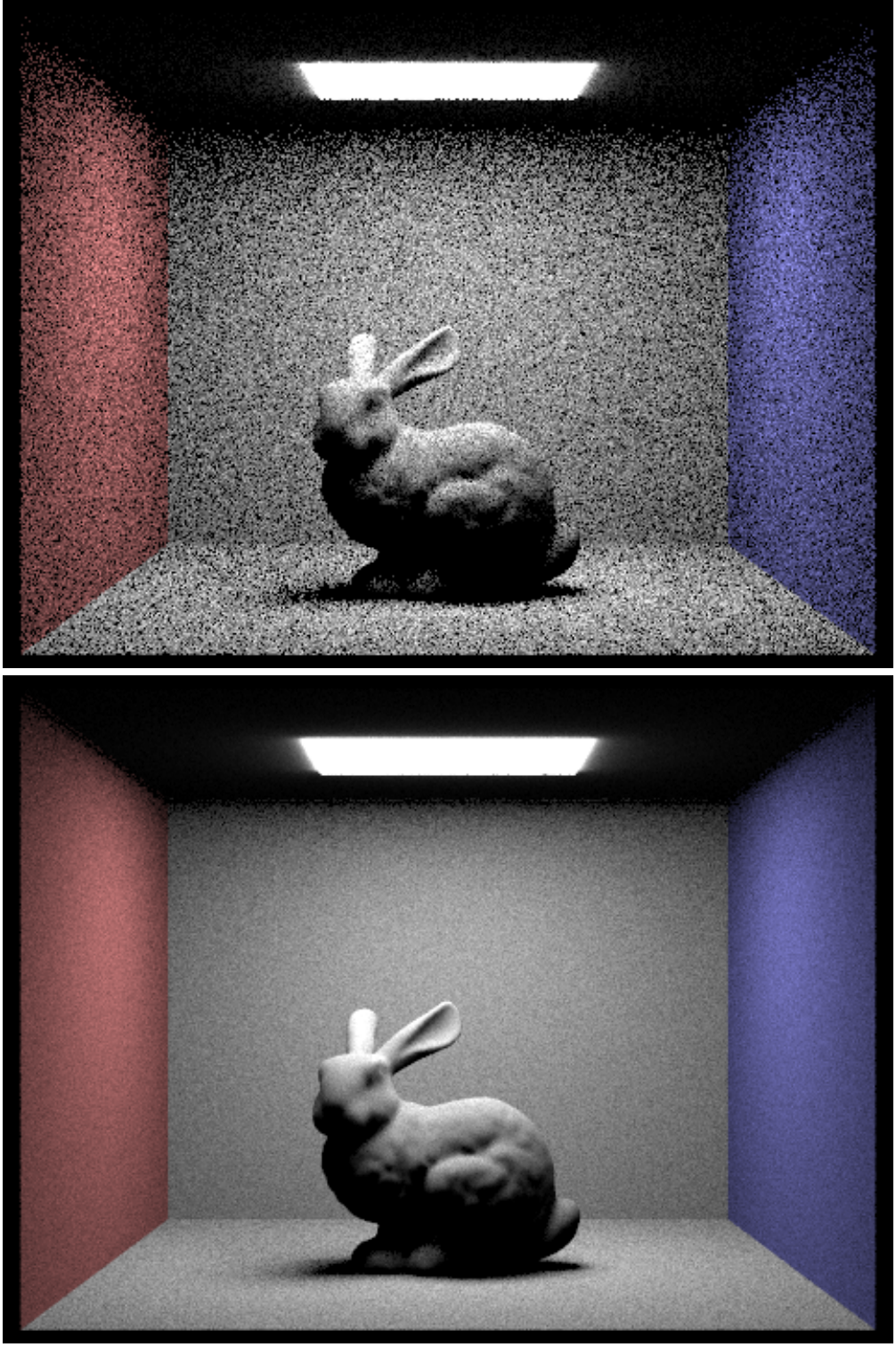
Direct Lighting with Uniform Hemisphere Sampling

Moving on to `PathTracer::estimate_direct_lighting_hemisphere(...)`, here we actually need to calculate the light reflected back towards the camera. We can do this by summing the rendering equation over every single direction. However, this is impossible, so we turned to a more feasible method: uniformly sample directions in the hemisphere, and only sum the rendering equation over those directions. My implementation in detail is as followed.

First, we specify the number of samples we are going to take and initialize a `Spectrum` variable *L_out*. Then we do the following for *num_samples* times:

1. Sample a direction using the sampler *hemisphereSampler*.
2. Since the direction returned by the sampler is in object space, we transform it to world space using the *o2w* matrix. The reason why we need to convert it to world space is because we are going to check for intersection with the primitives in our BVH, so we need to make sure the primitives and the ray are in the same space.
3. Then we make a *new_ray* using *hit_p* and world space *wi* direction.
4. Remember to set *new_ray*'s *min_t* to *EPS_F* to avoid numerical precision issue.
5. Initialize a new `Intersection` object *new_i*.
6. Test whether the new ray intersect with any primitives in our BVH.
7. If the ray does not hit any primitive, we move on to the next sample. If it does, we scale the radiance of the intersection by the amount of light that is been transferred from the new incoming direction to the camera direction, namely, the current BRDF from *wi* to *w_out*. And cosine of the angle of *wi*. Add this term to *L_out*.

After all the samples have been sampled, we normalize the result by dividing *L_out* by *num_samples* and *pdf*. Note that we are sampling from unit hemisphere, so the pdf is a constant 1/2π.



Left:render with 16 rays per pixel, 8 samples per area light; right: render with 64 rays per pixel, 32 samples per area light.

Direct Lighting by Importance Sampling Lights

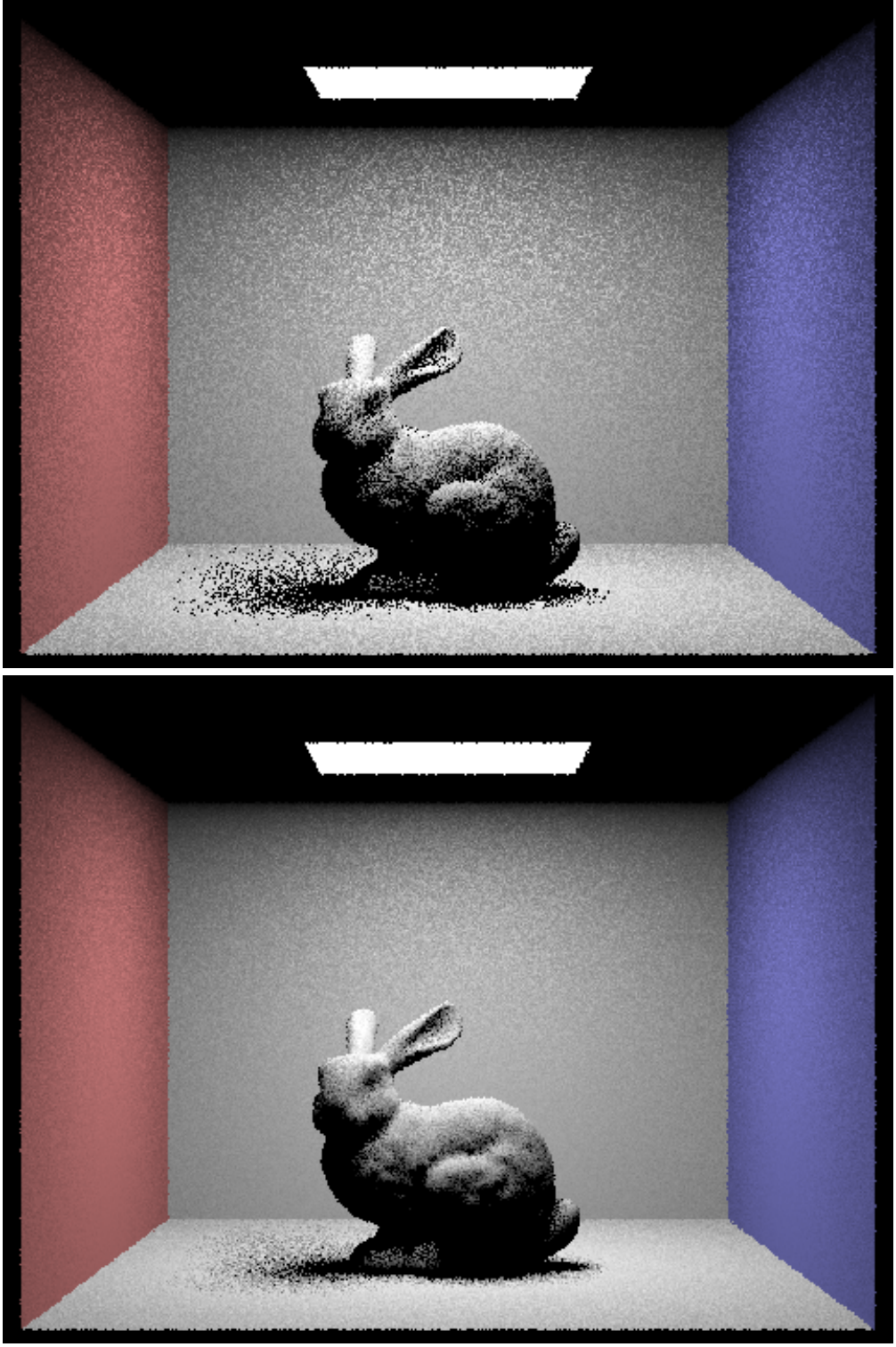
One issue with uniform hemisphere sampling, as we can see from the figures above, is the noisiness. This is because when we sample direction uniformly, it's inevitable to sample some directions that pointed away from the light source. When such ray is casted, it contributes nothing to the overall radiance *L_out*. Here we are going to sample the lights directly. We make sure that all the rays casted are in the direction toward some light. In this case, unless the ray is blocked by other surface, the sample should always return some radiance.

The implementation of `PathTracer::estimate_direct_lighting_importance(...)` goes as followed.

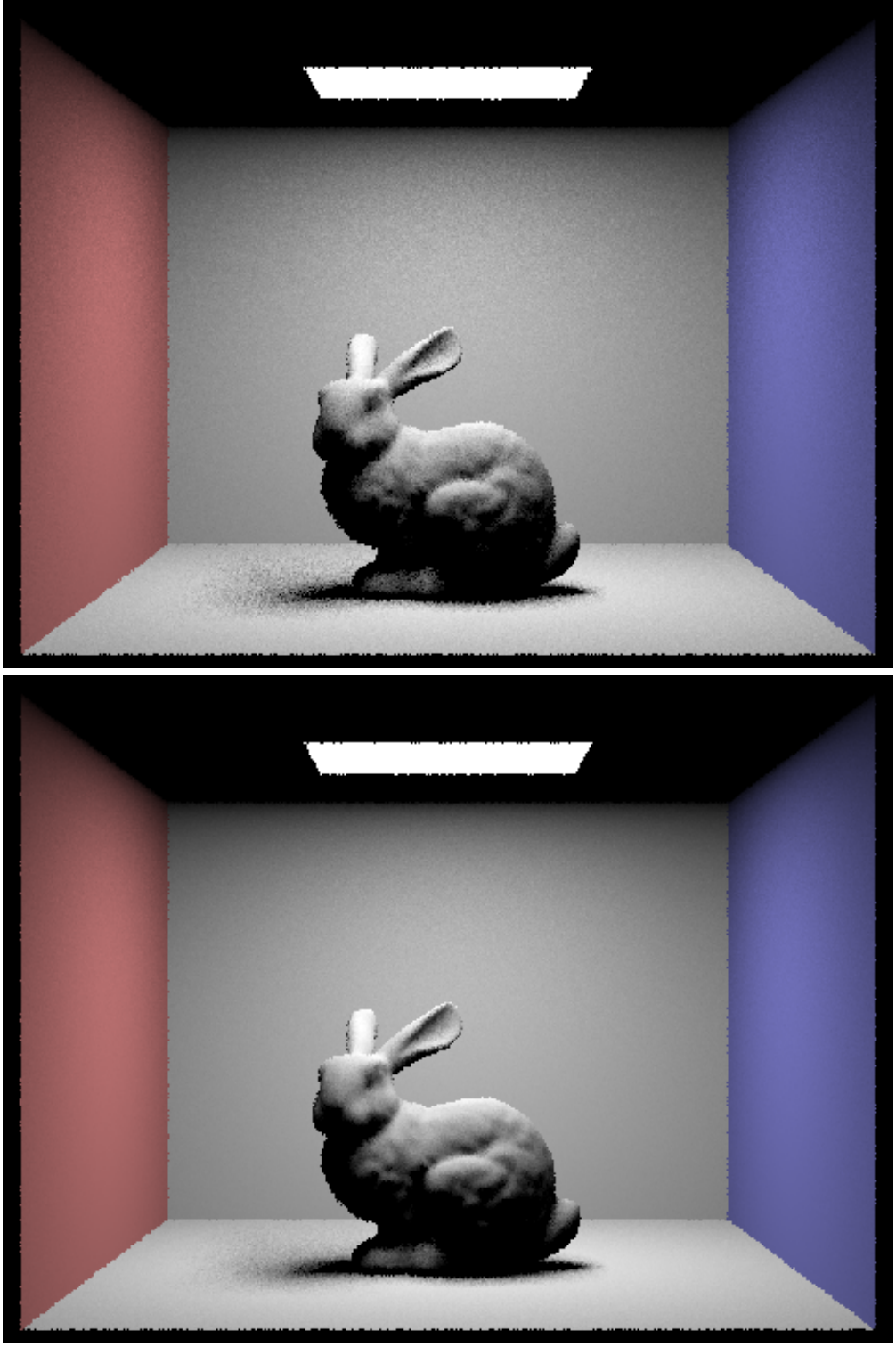
For each light in the scene:

1. If the light is a point light source, we only sample once, otherwise, we sample *ns_area_light* times.
2. Sample the light using `SceneLight::sample_L(...)`, also pass in pointers to get the incoming direction *wi*, the distance to light *dist*, and the *pdf* of sampling that direction.
3. Since the direction sampled by `sample_L` is in world space, convert it to object space by matrix multiply by *w2o*.
4. We do not want to sample any light source that is behind the hit point, therefore, if the z-coordinate of *wi* in object space is negative, we skip this sample.
5. 5Otherwise, we make a new ray with *hit_p* and *wi*, set the ray's *min_t* to *EPS_F*.
6. 6We make a new `Intersection` object *new_i*.
7. We check if this ray hits any thing before it hits the light, if it does, then the light will be blocked and we can skip the rest of this iteration. If the ray is not blocked, then we scale the *radiance* returned from `sample_L` by same factor as in the uniform hemisphere sampling case, but this time we normalize by dividing by the *pdf* and *ns_area_light*.

In the end, return *L_out*.

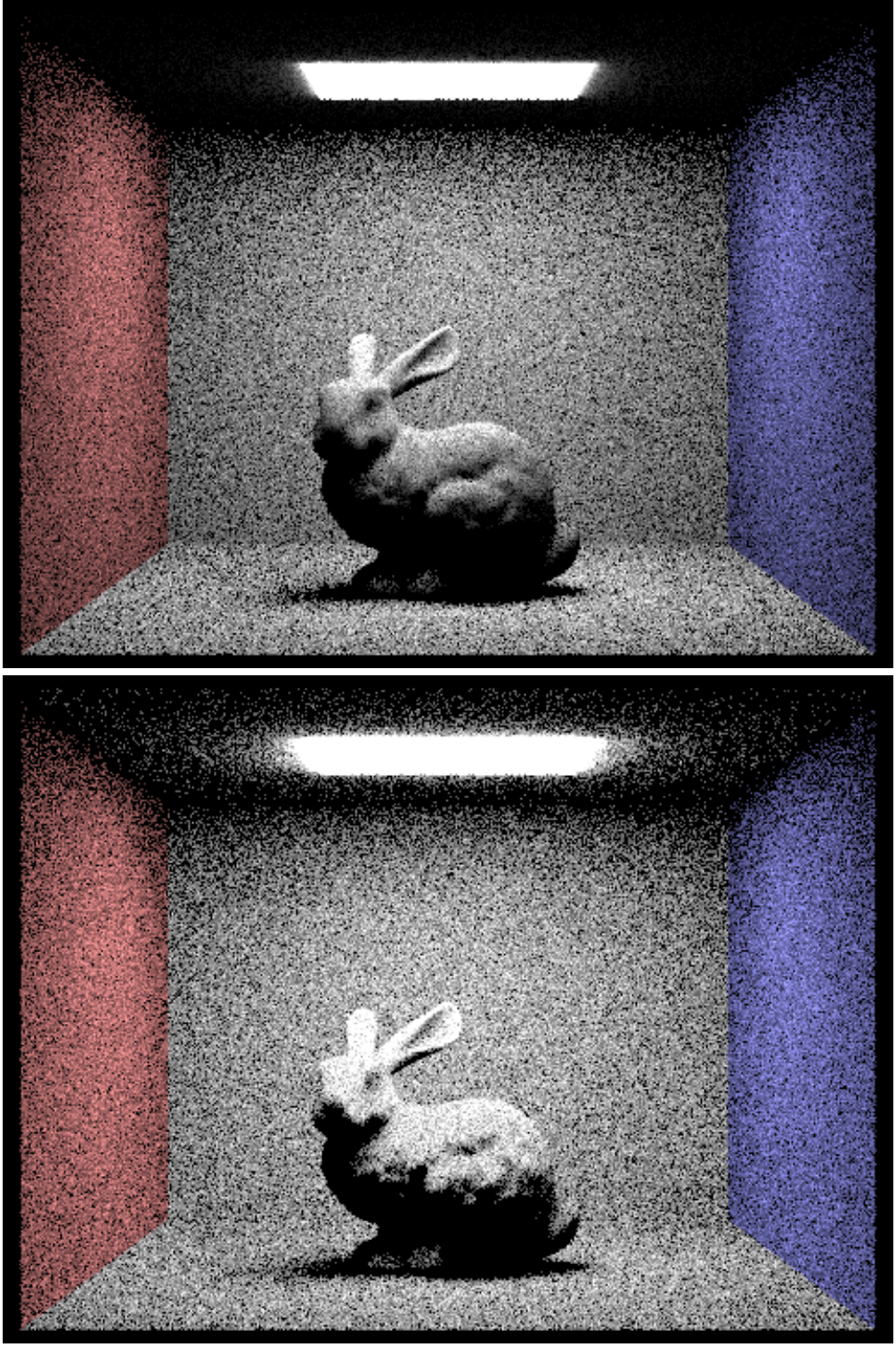


Left: render with 1 light ray, 1 sample per pixel; right: render with 4 light rays, 1 sample per pixel.

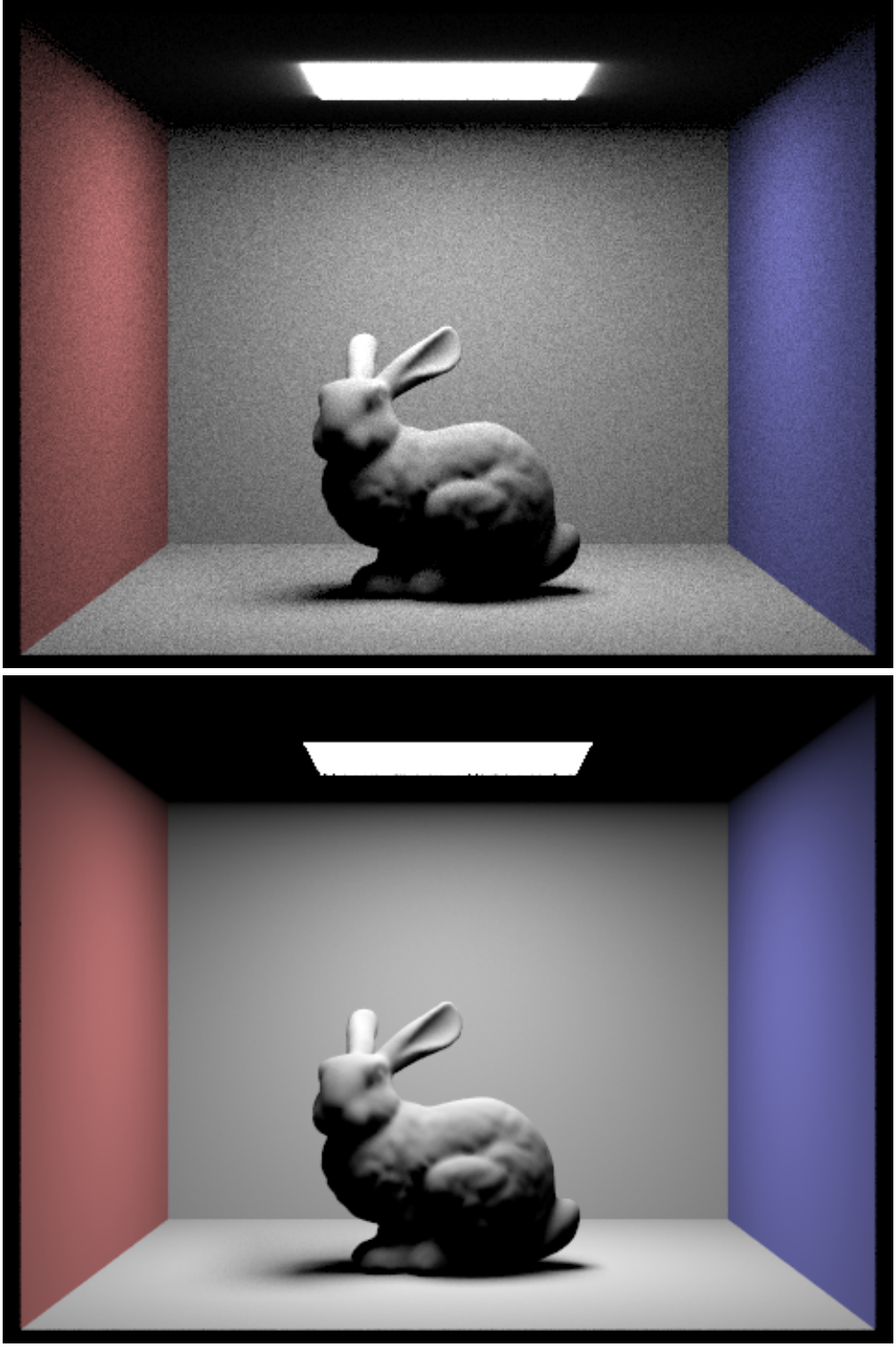


Left: render with 16 light rays, 1 sample per pixel; right: render with 64 light rays, 1 sample per pixel.

Let's compare the results between uniform hemisphere sampling and lighting sampling. See the side-by-side image comparion below.



Left: render with 1 light ray, 1 sample per pixel; right: render with 4 light rays, 1 sample per pixel.



Left: render with 16 light rays, 1 sample per pixel; right: render with 64 light rays, 1 sample per pixel.

We can't really compare the noisiness in the first pair of images since they are both pretty noisy. However, what we can compare is the brightness of the images. The image produced by direct light sampling is visibly brighter than the one produced by uniform hemisphere sampling. This result is reasonable since as I stated earlier, the direct light sampling method cast ray directly to the light sources, while uniform hemisphere cast ray in random direction, thus ray casted by uniform hemisphere sampling is more likely to hit a another dark surface. We can see difference in noisiness in the second pair of images. The image on the right is obviously smoother than the image on the left. This happens for similar reason as I stated above. Since direct light sampling always cast a ray toward the light, the radiance it gets should be more similiar and consistent compare to uniform hemisphere sampling.

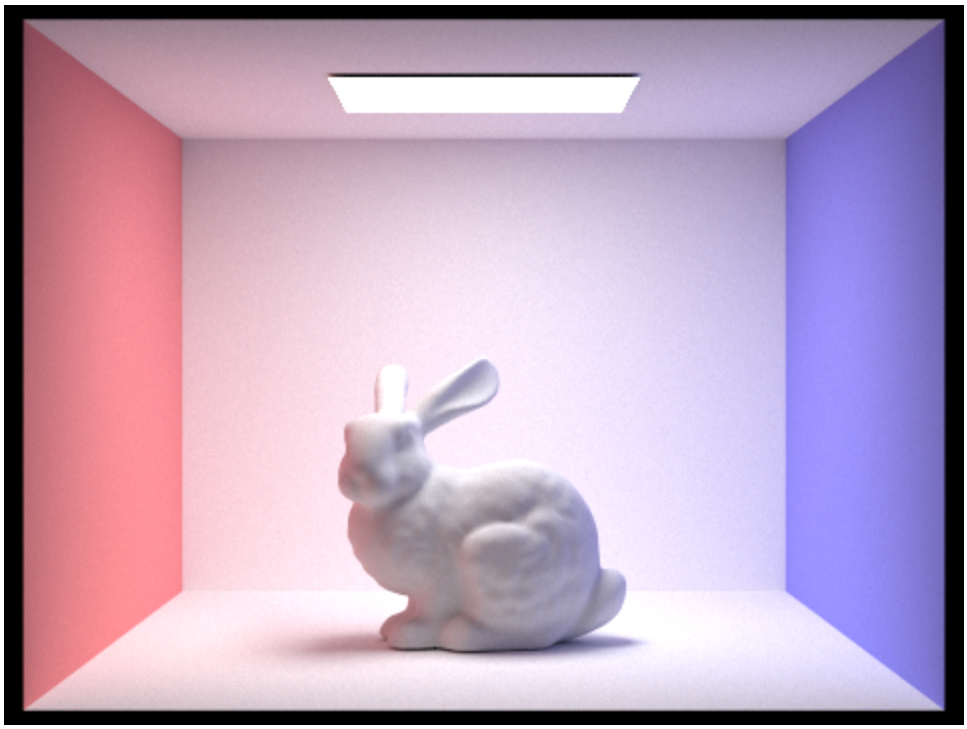
Part 4: Global Illumination

In previous parts, the ray bounces at most once. Now we are going to implement the function `PathTracer::at_least_once_bounce_radiance(...)` to trace light that bounces multiple times. This function is going to call itself recursively, the stopping condition is when the depth of the ray is at most one, or the Russian Roulette stops with some probability, or that the ray does not intersect the scene. While these conditions are not met, we do the following:

1. We call `one_bounce_radiance(...)` we coded in last part to get the radiance of the current bounce, we call this *L_out*.
2. We check if the *max_ray_depth* has been reached, if it is, we return *L_out*. We also return *L_out* if the *max_ray_depth* has not been reached but the Russian Roulette decides to terminate with continuation probability 0.6.
3. The recursion is not terminated, we continue to the next bounce. We first find the hit point and outgoing direction of the current intersection result from the function call we made at the beginning of the loop.
4. Remember the `sample_f` function we wrote in part 1? Now it is the time we use it to sample an incoming direction. As always, we need two version of the incoming direction, one in the world space, one in the object space. World space for ray intersection with the primitive, object space to use in current intersection's bsdf function to get the outgoing reflection.
5. We create a new Ray object with the hit point as origin and the sampled incoming direction as direction. Aside from setting *min_t* to *EPS_F*, this time we need to also set the new ray's depth to the current depth minus 1.
6. We create a new `Intersection` object. We check if any primitive in our BVH intersects the new ray.
7. If there is an intersection, we call `at_least_one_bounce_radiance` recursively, scale it by the outgoing reflection of the current intersection and cosine of the incoming direction angle just like we did before. However, this time, besides dividing by the pdf of *w_i*, we also divide by the pdf of the Russian Roulette, 0.6, to normalize.

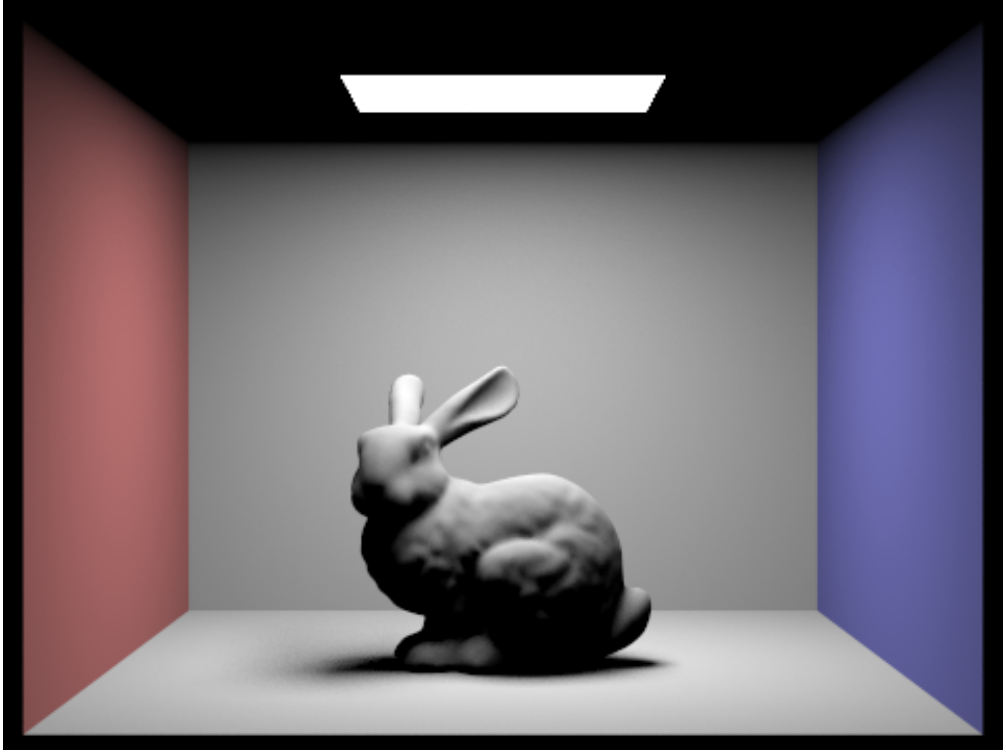
In the end, *L_out* is returned.

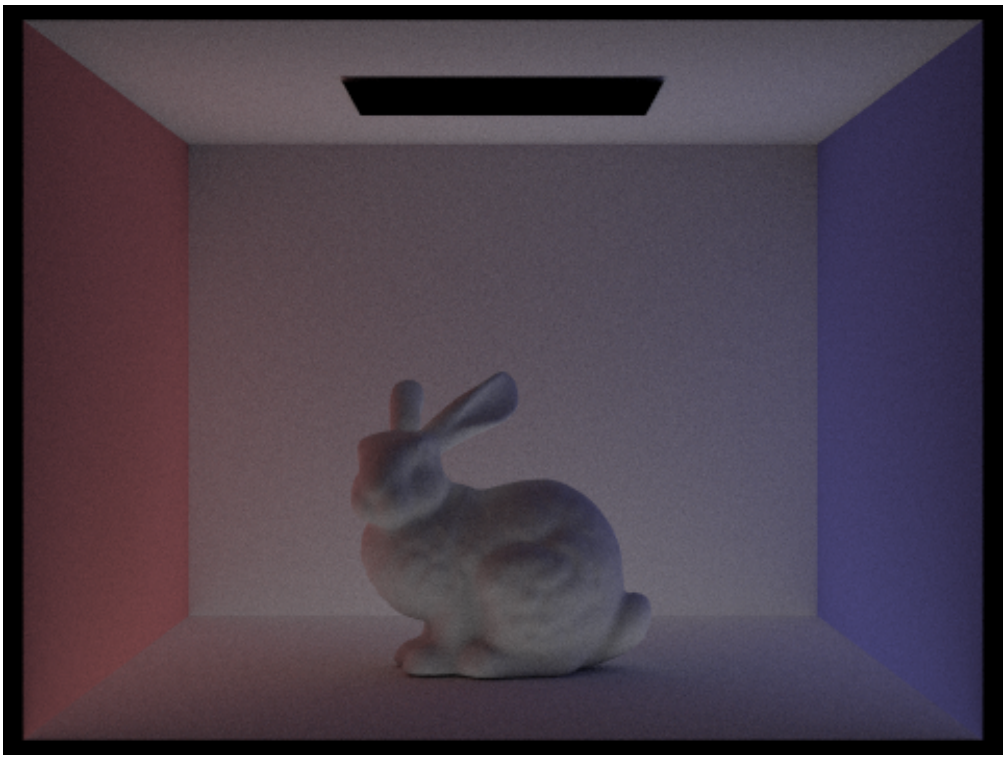
Below is an image render with both direct and indirect light.



CBbunny.dae render with 1024 samples per pixel.

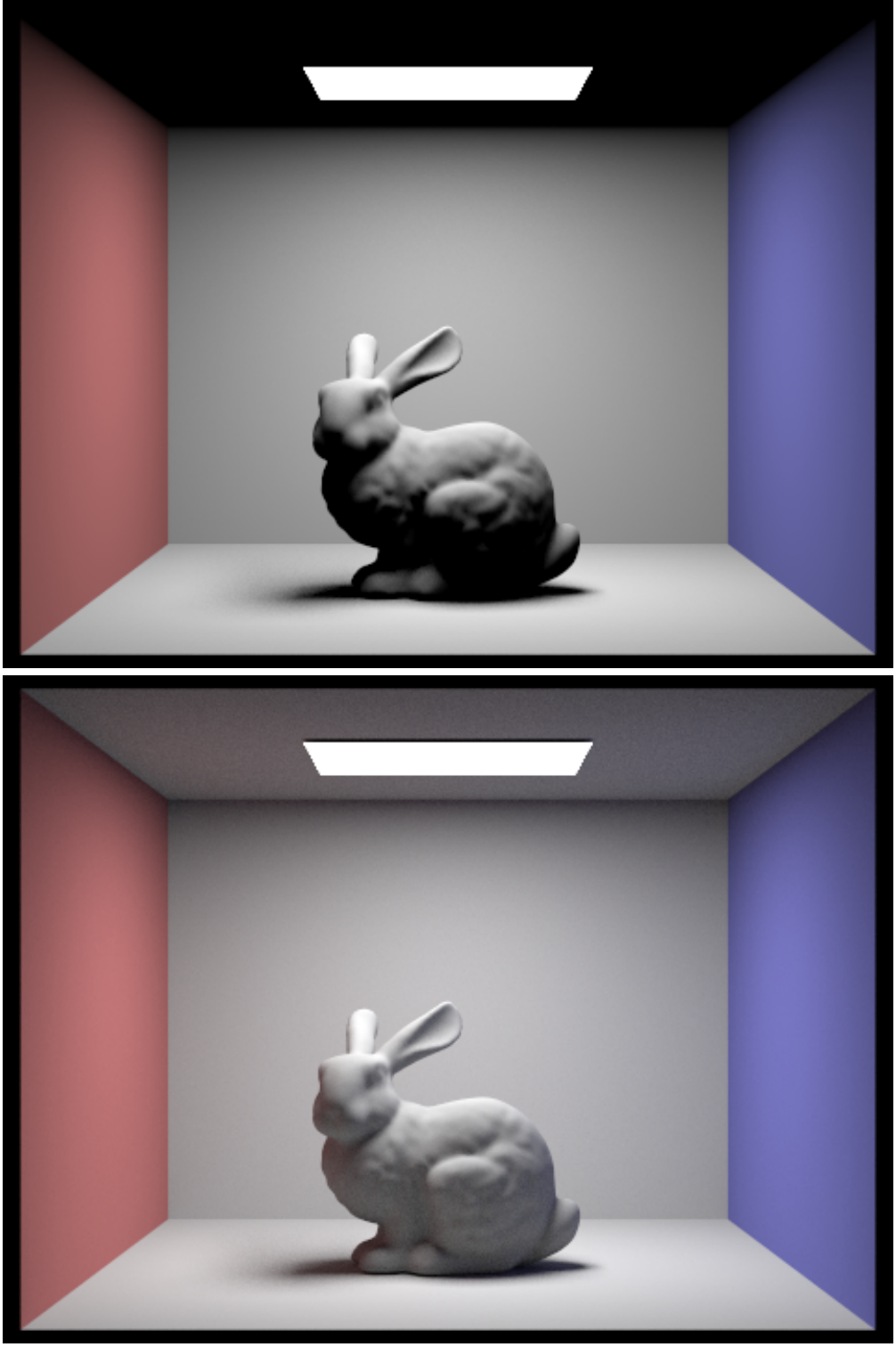
Let's see the results of rendering with only direct light and only indirect light.



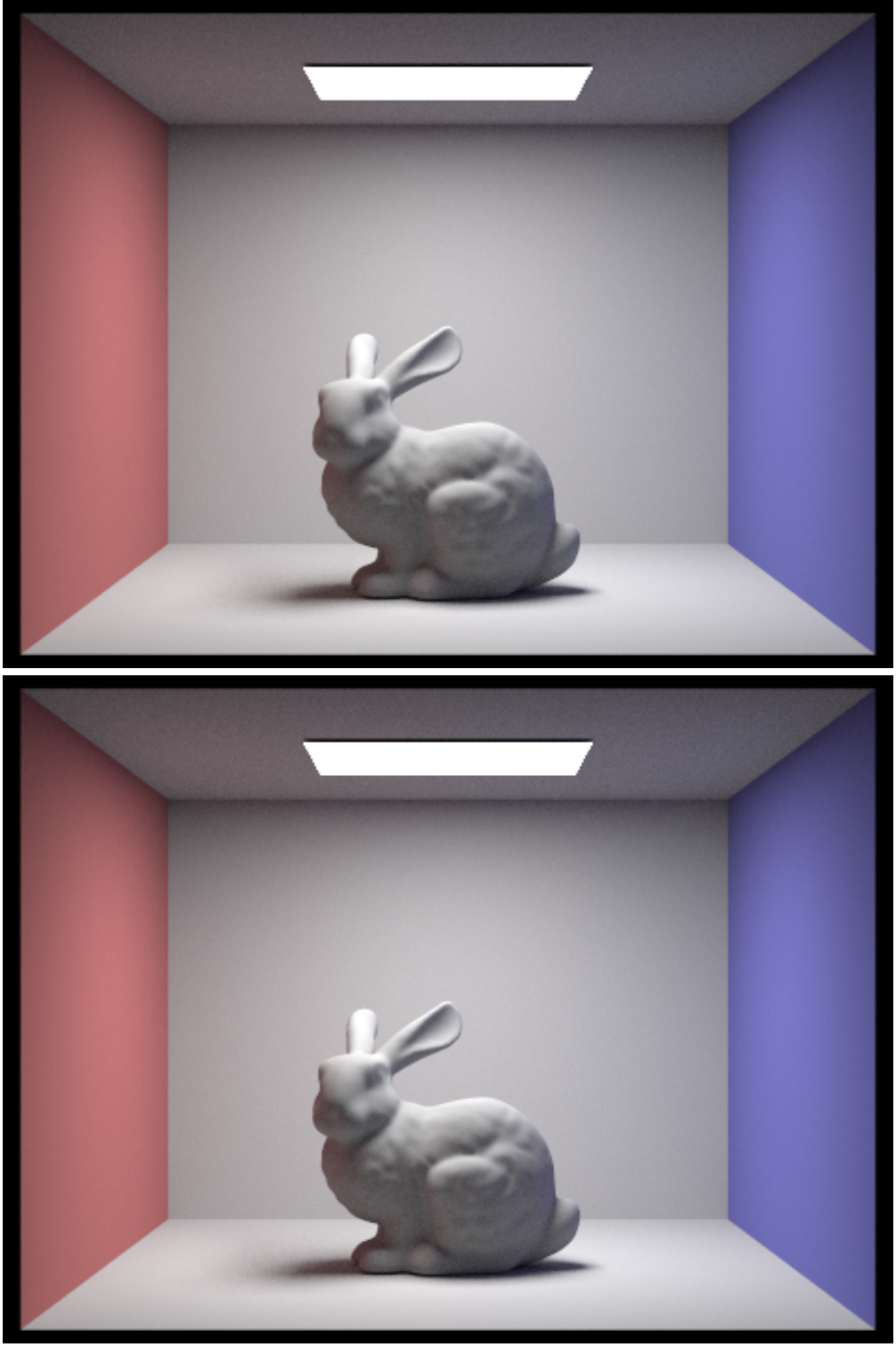


Left:render with 1024 rays per pixel, 1 samples per area light, only direct light; right: render with 1024 rays per pixel, 1 samples per area light, only indirect light.

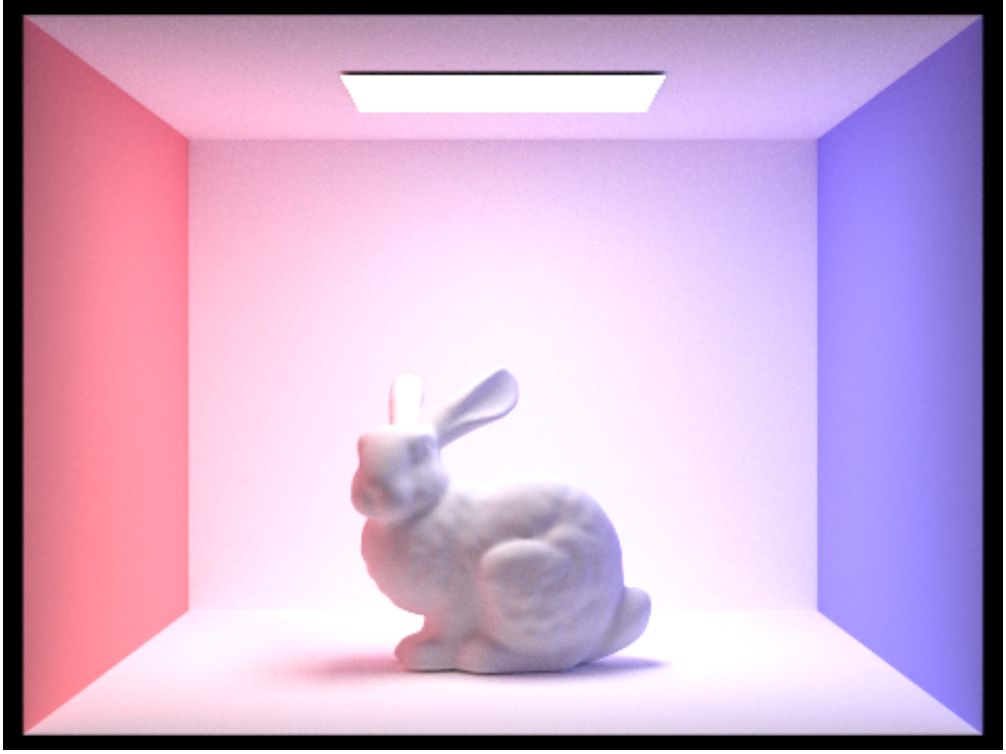
The five imgaes below compre rendered views with *max_ray_depth* set to 0, 1, 2, 3, and 100.



Left: render with 1024 samples per pixel, max_ray_depth 0; right: render with 1024 samples per pixel, max_ray_depth 1.

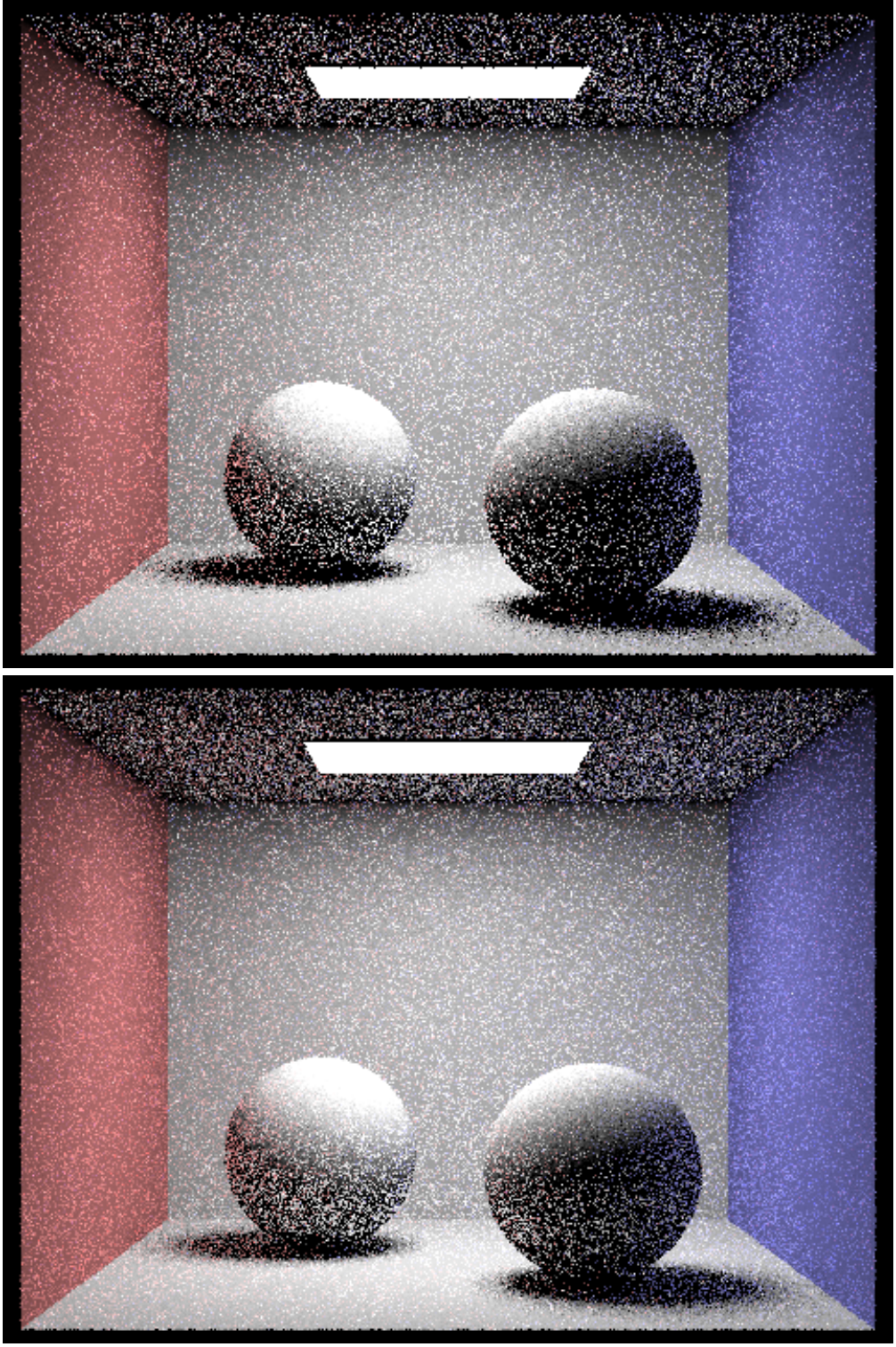


Left: render with 1024 samples per pixel, max_ray_depth 2; right: render with 1024 samples per pixel, max_ray_depth 3.

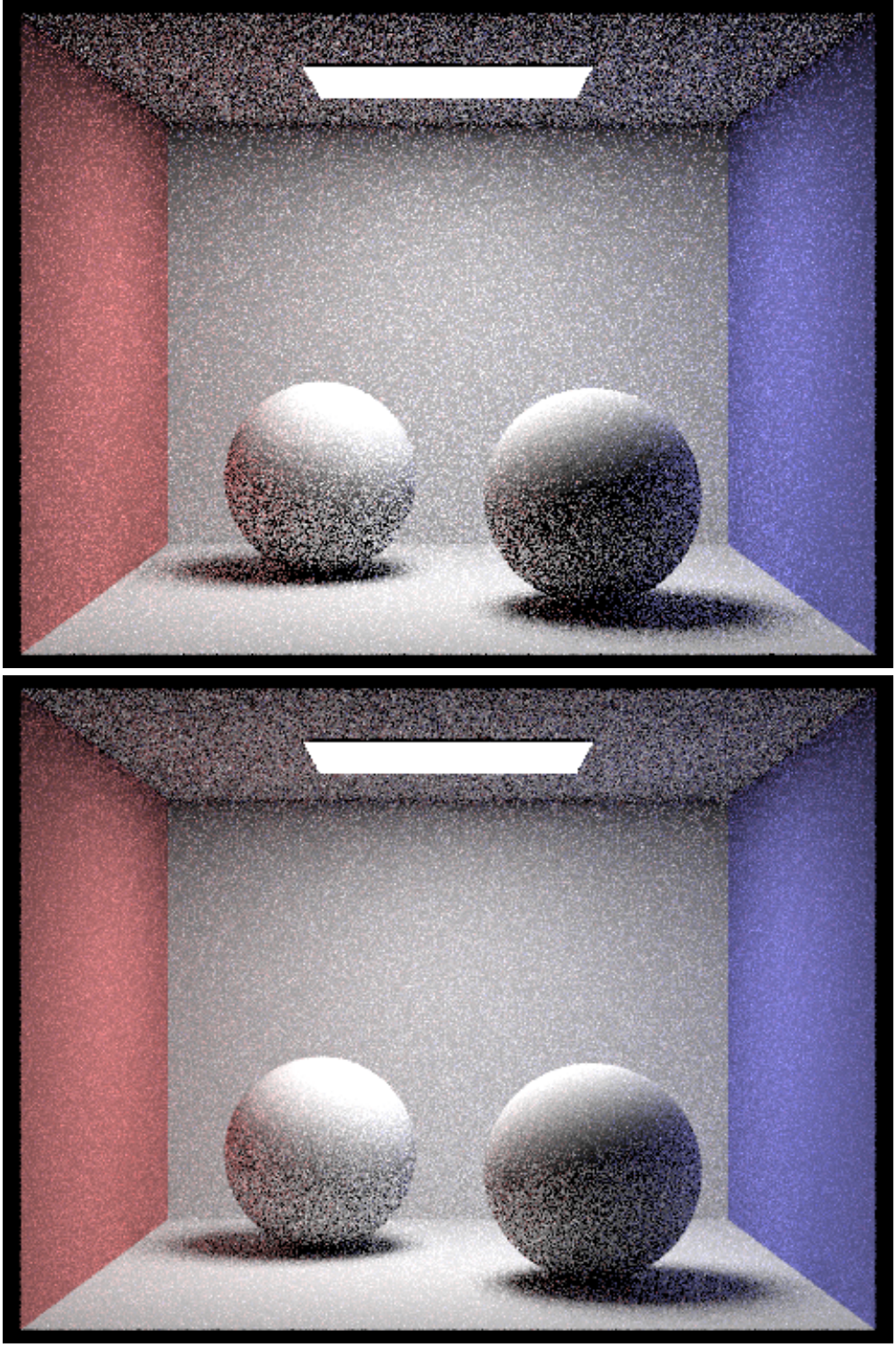


Render with 1024 samples per pixel, max_ray_depth 100.

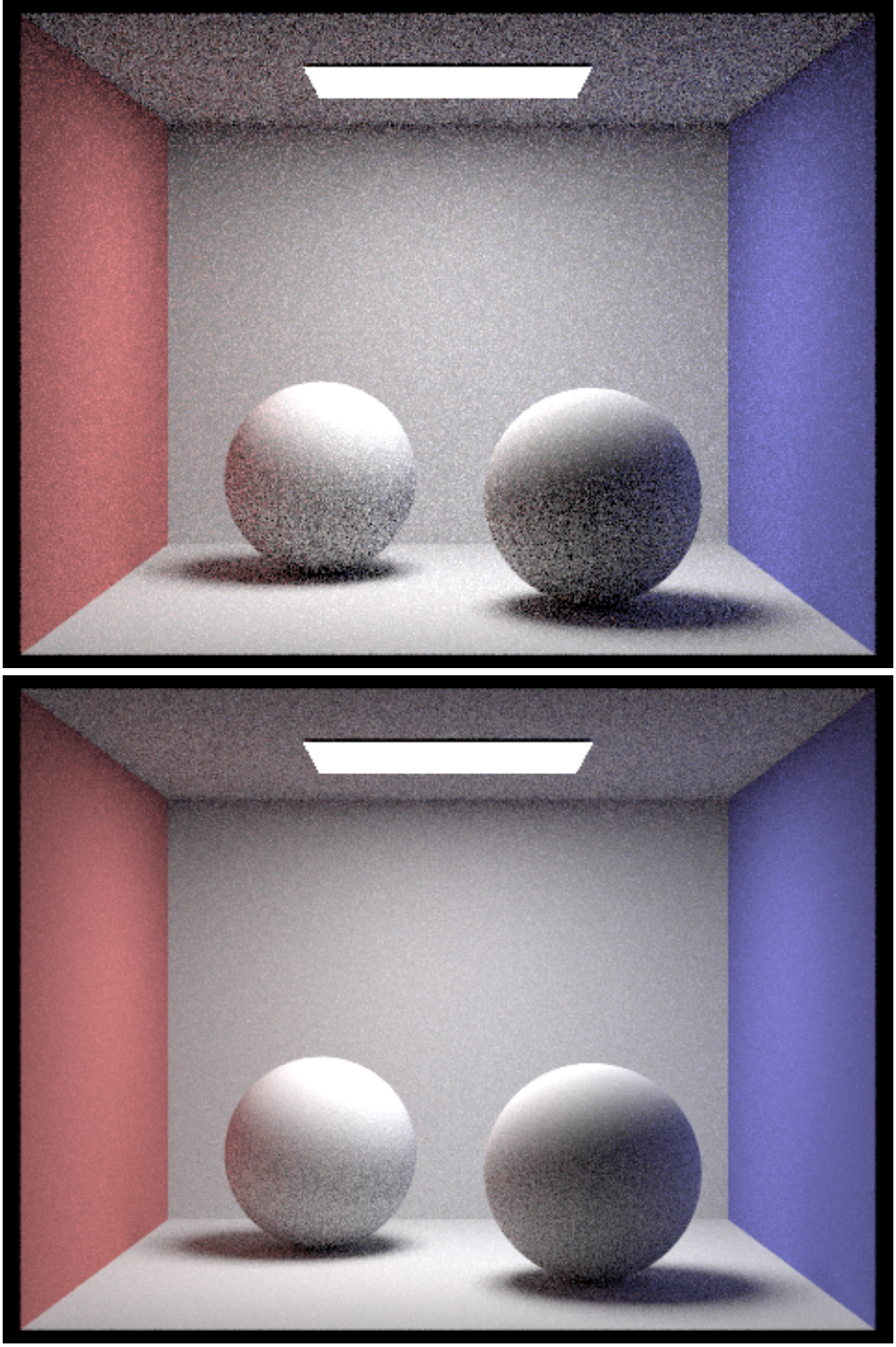
Now, let's compare *CBspheres.dae* rendered with 1, 2, 3, 4, 8, 16, 64, and 1024 sample-per-pixel rates.



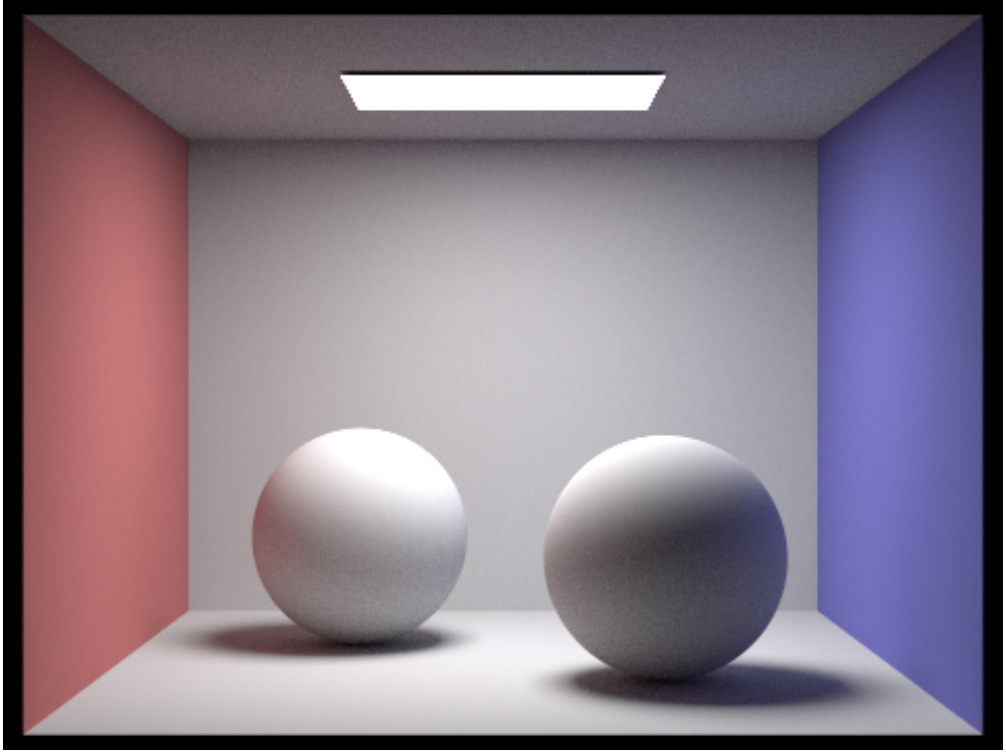
Left: render with 1 sample-per-pixel rate, 4 light rays; right: render with 2 sample-per-pixel rate, 4 light rays.



Left: render with 4 sample-per-pixel rate, 4 light rays; right: render with 8 sample-per-pixel rate, 4 light rays.

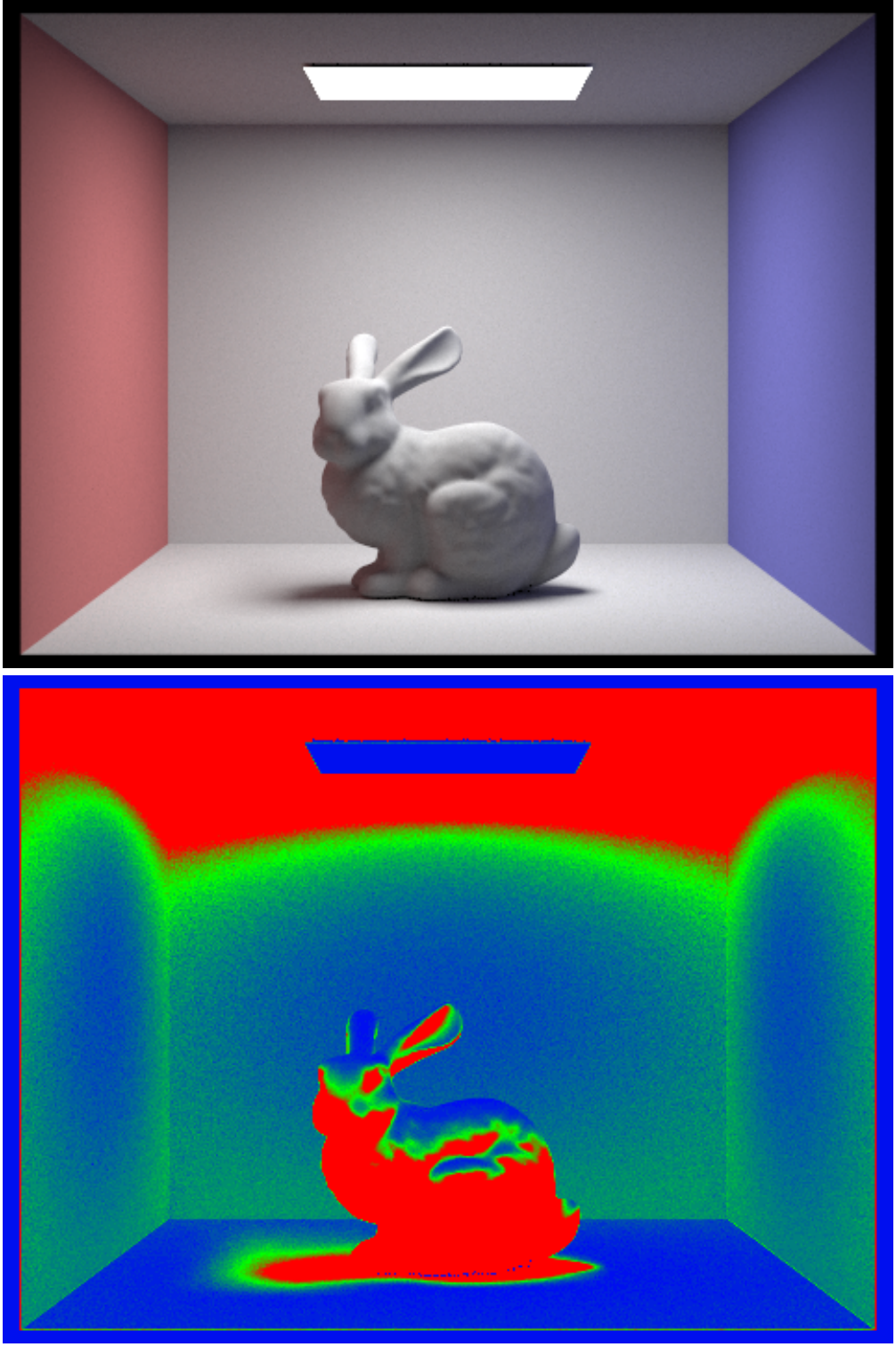


Left: render with 16 sample-per-pixel rate, 4 light rays; right: render with 64 sample-per-pixel rate, 4 light rays.



Render with 1024 sample-per-pixel rate, 4 light rays.

Part 5: Adaptive Sampling



Left: CBbunny.dae render with 2048 samples per pixel; right: sample rate image.

The implementation of adaptive sampling is rather straight forward. We need to keep track of two values *s1* and *s2*, and use them to check if the condition $I \leq maxTolerance \cdot \mu$ at the right time. With that said, I first created two float variable *s1* and *s2* and initialize them to zero. Then, I changed the loop setting from `(int i = 0; i < num_samples; i++)` to `(int i = 1; i <= num_samples; i++)`. This is because we need the correct number of samples taken every time we enter the loop. In part 1 we only need to incorporate the radiance returned by `est_radiance_global_illumation(...)` to the Monte Carlo estimator, now we need to take one more step and get its illumination, add it to *s1*, and the square of it to *s2*. Moving on, if the current number of samples we have taken is a multiple of *samplesPerBatch*, we calculate the *mu* and *sigma2* using *s1* and *s2*. We check if the condition above is satisfied. If it is, we say that the Monte Carlo estimator has converged, so divide it by the current number of samples taken and update the *sampleBuffer*. We also set the *sampleCountBuffer* to the current *i* value and return.