Sam Zhou

CS184 Project 3-1 : Pathtracer

The goal of this project is to create a tool that renders scenes using raytracing techniques. For this project we will only use diffuse surfaces and will instead focus on calculating light trajectories and different illumination techniques.

Part 1 - Ray Generation and Scene Intersection

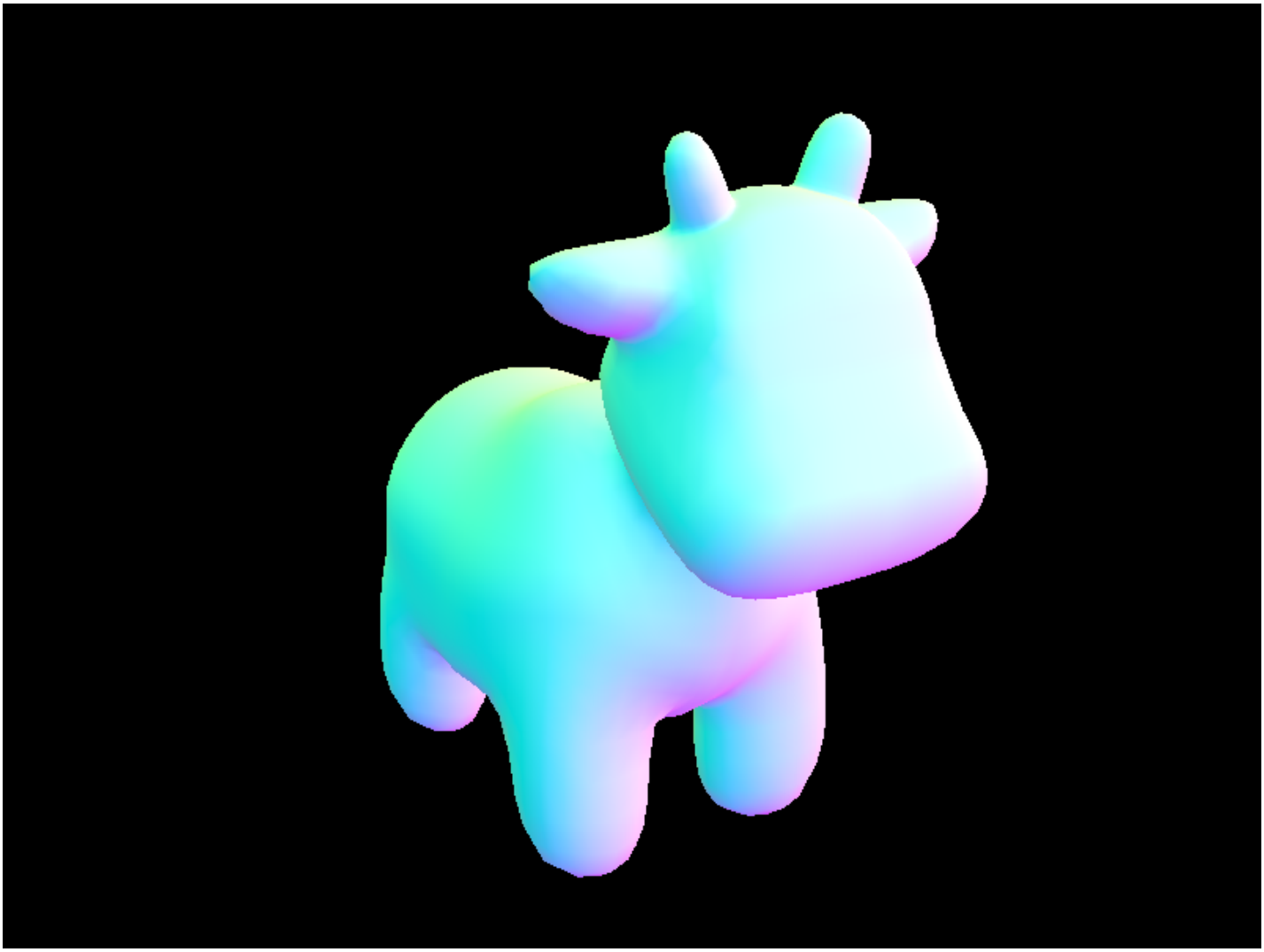
The idea of ray tracing is to start from the perspective of a camera and follow a path from the camera to the scene in question. For each pixel of the image, we trace some number of light rays and check if this light ray intersects with the objects in our scene. If they do, we can calculate the amount and color of the light that travels from that intersection point to the camera.

First, we will just focus on checking intersections with the primitives in our scene. Since there is no way to determine beforehand which primitives our ray intersects with and which of all these the ray hits first, we need to iterate through all of the primitives in our scene and keep track of which primitive the ray intersects with first. This is done easily by updating the max range of the ray whenever we find an earlier intersecting primitive. If we do this on every intersection, the last intersection we find that is within the max range of our ray will be the closest.

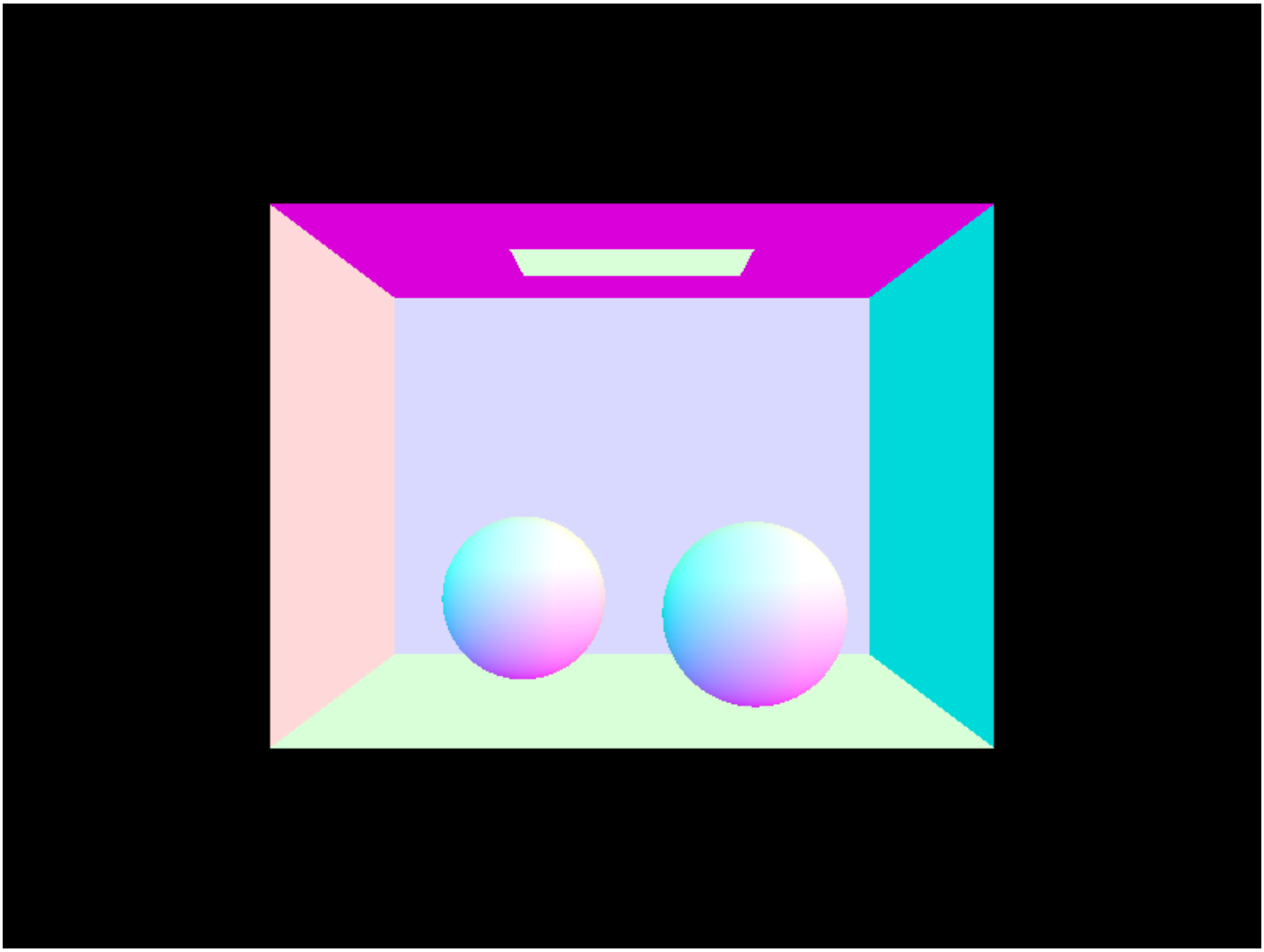
For triangle primitives, we can check intersection by making sure that the barycentric coordinates for all three vertices are between 0 and 1. If this is true, then the combination of these coordinates and vertices will be somewhere inside of the triangle. We can calculate the barycentric coordinates using dot products between the ray and the vertices of the triangle as described in lecture.

Here are some examples of what we can render just after implementing basic ray tracing and scene intersection with the use of normal vector shading.

Moo Moo



Some Spheres That Will Appear A Lot



Part 2 - Bounding Volume Hierarchy

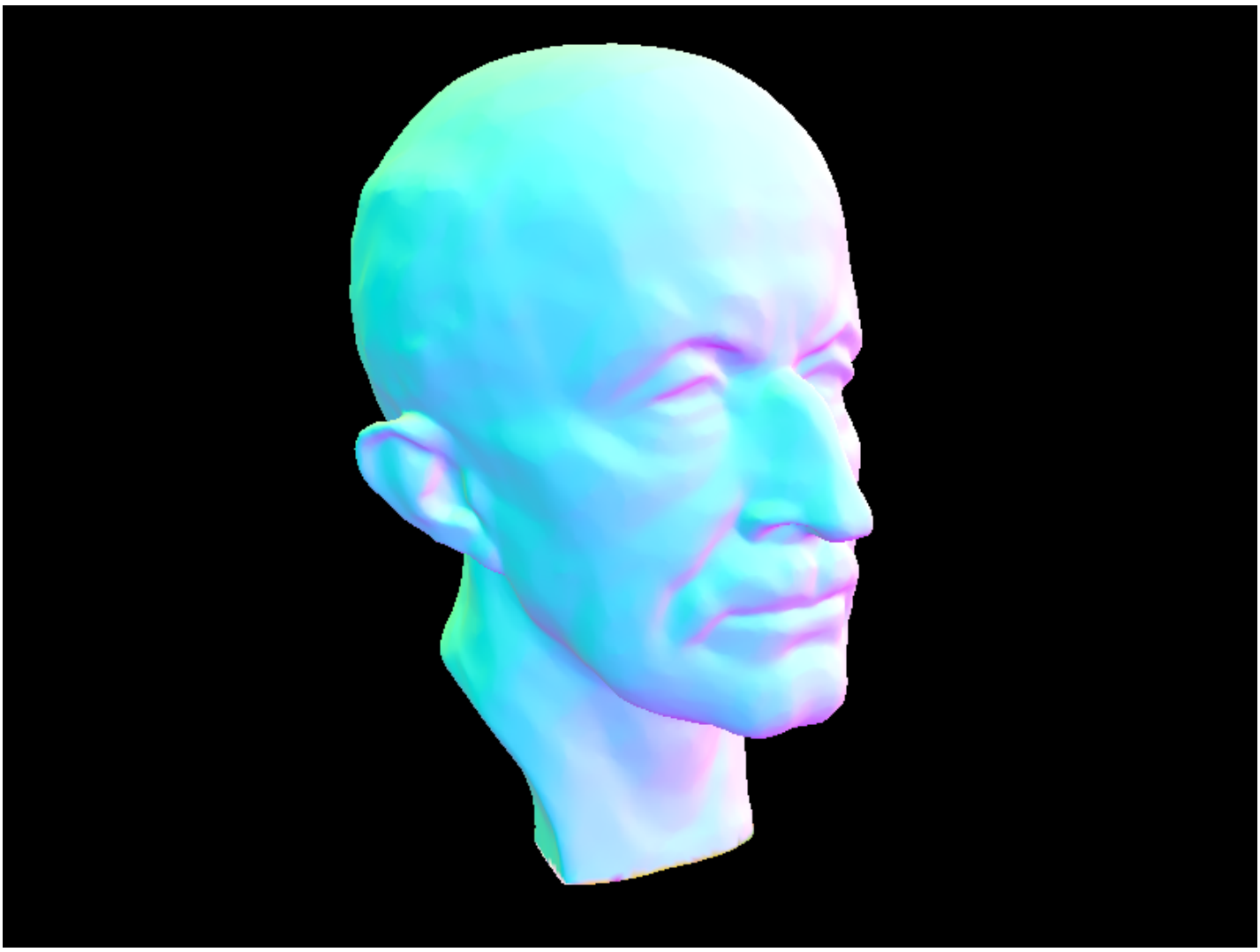
The issue with the current intersection algorithm is that it is horribly slow to iterate over every single primitive and check intersections with all of them. Instead, we will implement a Bounding Volume Hierarchy to speed up this process. A BVH is basically a binary search tree where every node represents some 3D bounding box in our scene which contains some set of primitives. This will make it easier to narrow down our search to only the relevant primitives.

First, to construct our BVH we need to evenly split our primitives into nodes for our binary search tree. Given our initial set of primitives, we find the bounding box for all of them and look at the largest dimension. We use a value along this dimension this as our splitting point and partition our primitives into two subsets - one greater and one less than our splitting point on the chosen dimension. Then we can recursively split each subnode in our tree until we reach a point where the primitives remaining are small enough that we are comfortable iterating through the entire set. In terms of picking a splitting point on the chosen dimension, I used the midpoint of the centroid bounding box. I just found this to rather evenly split the primitives and usually did not leave any subset empty.

Now that we have our BVH, we need to update our intersection algorithm to use our BVH instead of iterating one through all of the primitives. Now we can traverse our BVH by first checking if the ray intersects the current node's bounding box. This is done by finding both the entering and exiting intersection times and checking that the max range of the ray has some overlap with this intersection range. If it does, we recurse on both children of the node. If the ray does not intersect the bounding box then we can just stop checking that subtree. Once we reach a leaf node, we can iterate through all of the primitives once more and calculate intersections with those. Overall, this saves a lot of intersection checks since we are narrowing down our search to just the primitives in a few leaf nodes.

With this improvement, we can render images with far more primitives than before. Here is one such example.

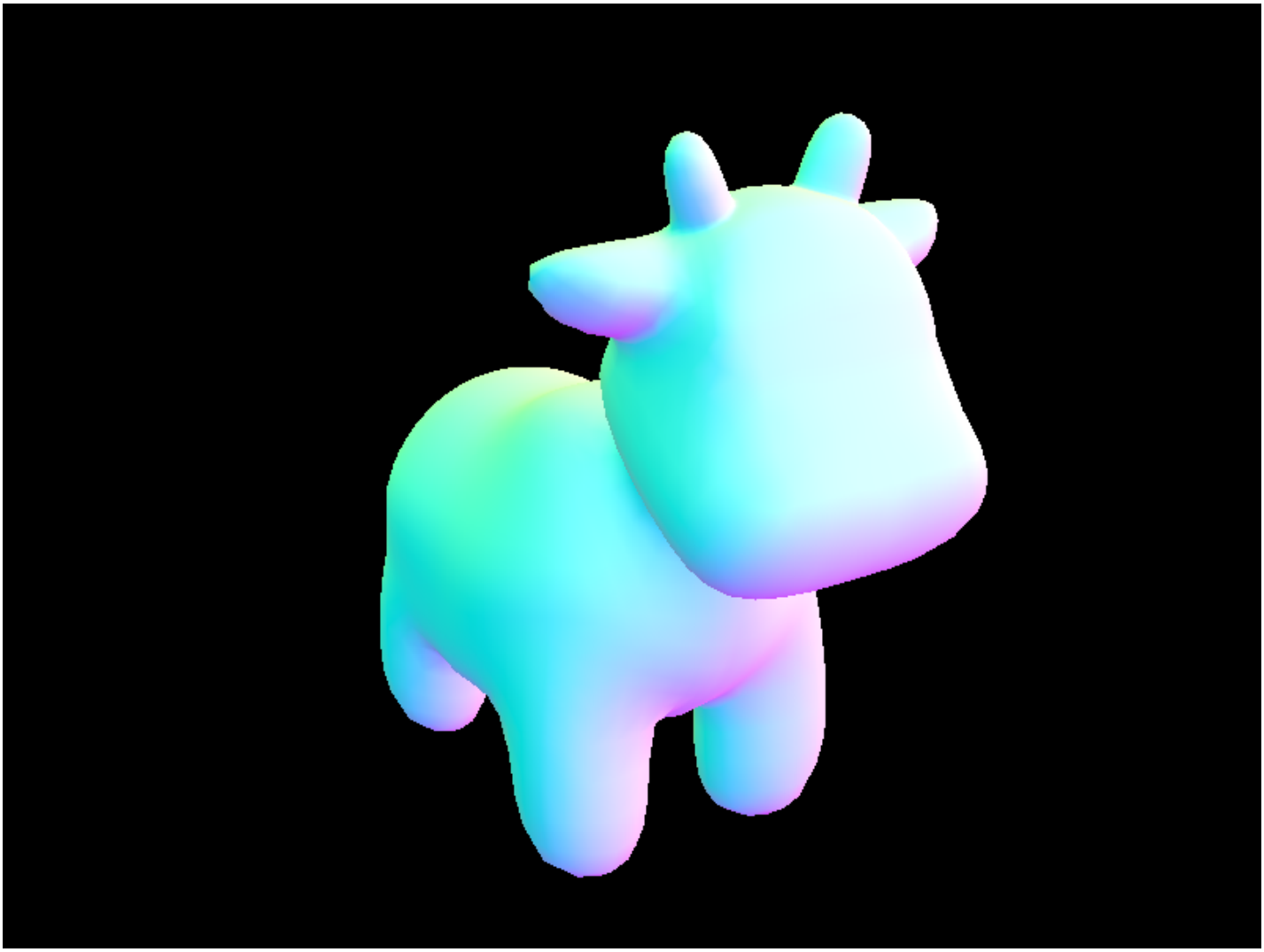
Mr Planck



BVH vs Naive

To get some sense of how much faster this is, we can look at the improvement on runtime after implementing our BVH. First we have some stats on the cow that was rendered in part 1.

Cow Stats



No BVH

```
[PathTracer] Input scene file: ../dae/meshedit/cow.dae
[PathTracer] Collecting primitives... Done! (0.0011 sec)
[PathTracer] Building BVH from 5856 primitives... Done! (0.0004 sec)
[PathTracer] Rendering... 100%! (43.2477s)
[PathTracer] BVH traced 430453 rays.
[PathTracer] Averaged 2485.467487 intersection tests per ray.
```

With BVH

```
[PathTracer] Input scene file: ../dae/meshedit/cow.dae
[PathTracer] Collecting primitives... Done! (0.0018 sec)
[PathTracer] Building BVH from 5856 primitives... Done! (0.0104 sec)
[PathTracer] Rendering... 100%! (0.3696s)
[PathTracer] BVH traced 480000 rays.
[PathTracer] Averaged 3.436060 intersection tests per ray.
```

You can see that without the BVH, our render took 43.25 seconds. After adding the BVH, it only took 0.37 seconds. This is a very significant increase and this object would not even be considered very complex. Here are some stats for a more complex figure.

Beast Stats

```
[PathTracer] Input scene file: ../dae/meshedit/beast.dae
[PathTracer] Collecting primitives... Done! (0.0080 sec)
[PathTracer] Building BVH from 64618 primitives... Done! (0.0977 sec)
[PathTracer] Rendering... 100%! (0.2549s)
[PathTracer] BVH traced 480000 rays.
[PathTracer] Averaged 2.079706 intersection tests per ray.
```

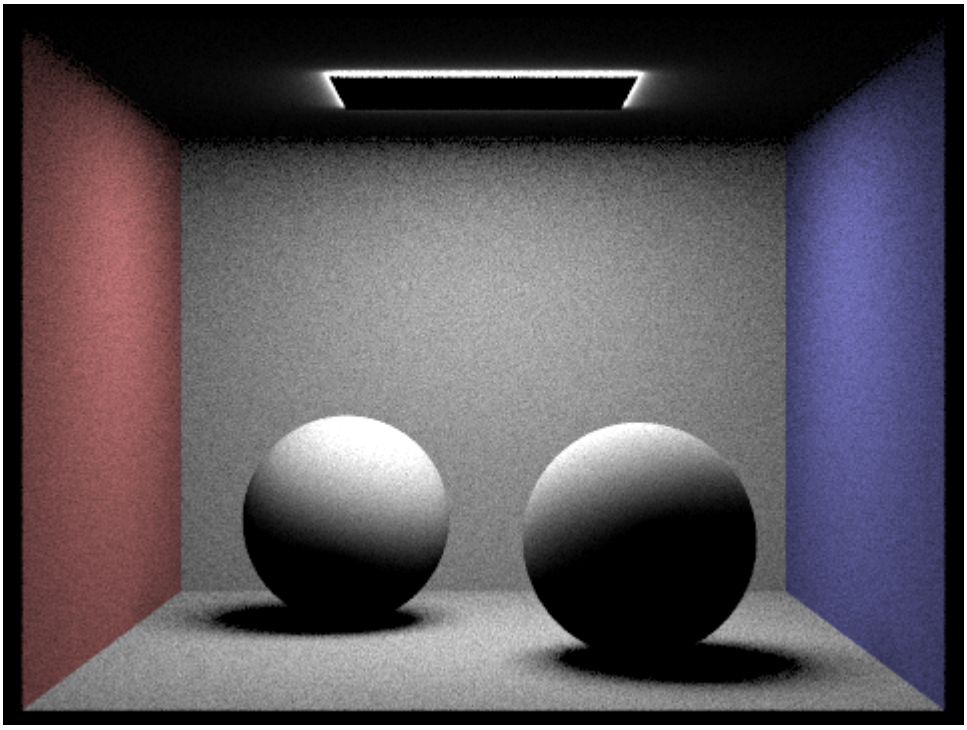
This one has 64618 primitives compared to the 5856 in the cow. The render only took 0.25 seconds which is on the same order of the cow even though the number of primitives is 10x the number in the cow.

Part 3 - Direct Illumination

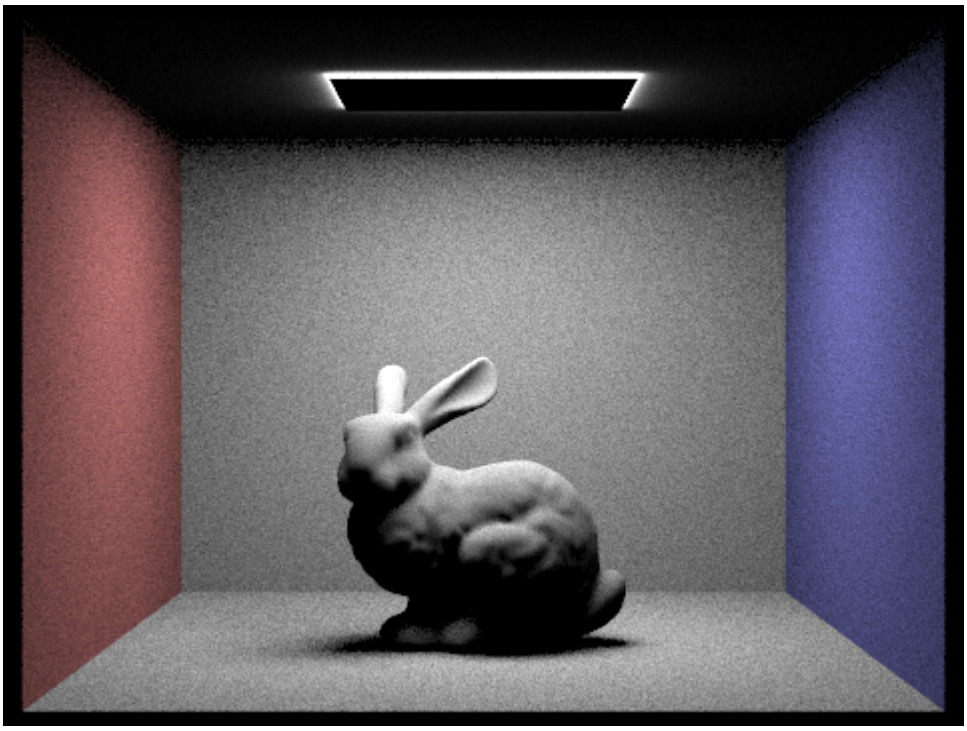
Now that we have a better implementation of checking ray intersection, we will try to improve our calculations for the light captured in these rays instead of using the normal vector as the color. The first step is to calculate direct illumination which is the amount of light coming directly from the light to the point of intersection of our ray.

First, we implemented hemisphere sampling. This is the idea of taking multiple samples uniformly randomly from the hemiphere aligned with the normal of the surface at the point of intersection. With the random ray, we extend it and check if it intersects with anything. If it does, we get the emission from that intersection which will only be non-black if it is a light. The idea is if we sample enough times, we are likely to find all the lights visible from that intersection point. Here are some of the results from hemisphere illumination.

Spheres again but hemispheres



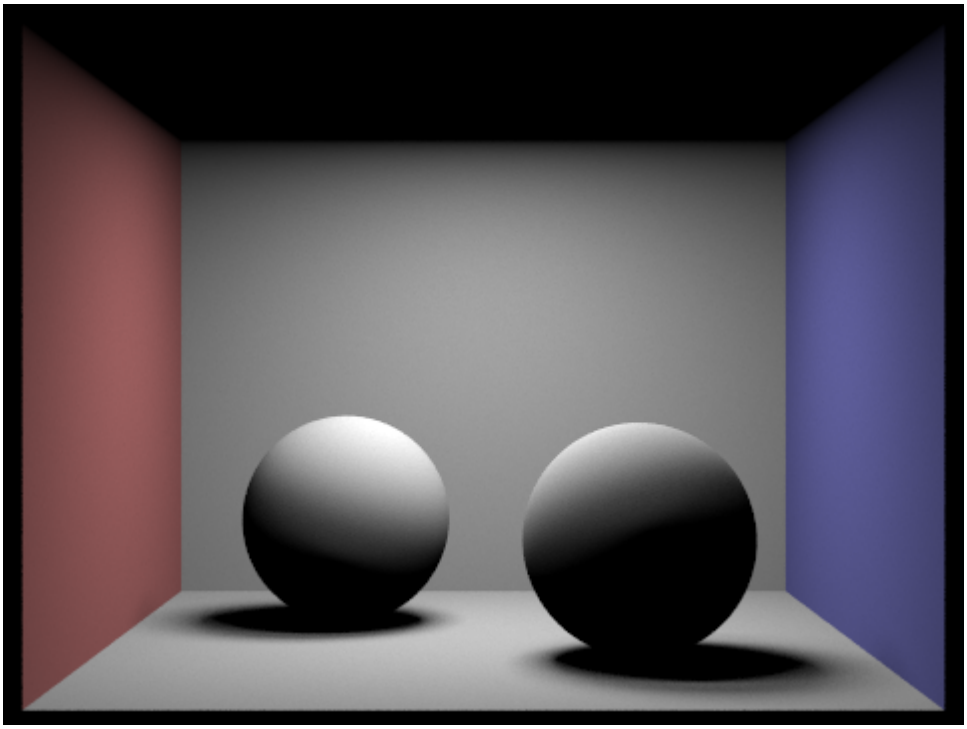
Hemisphere BunBun



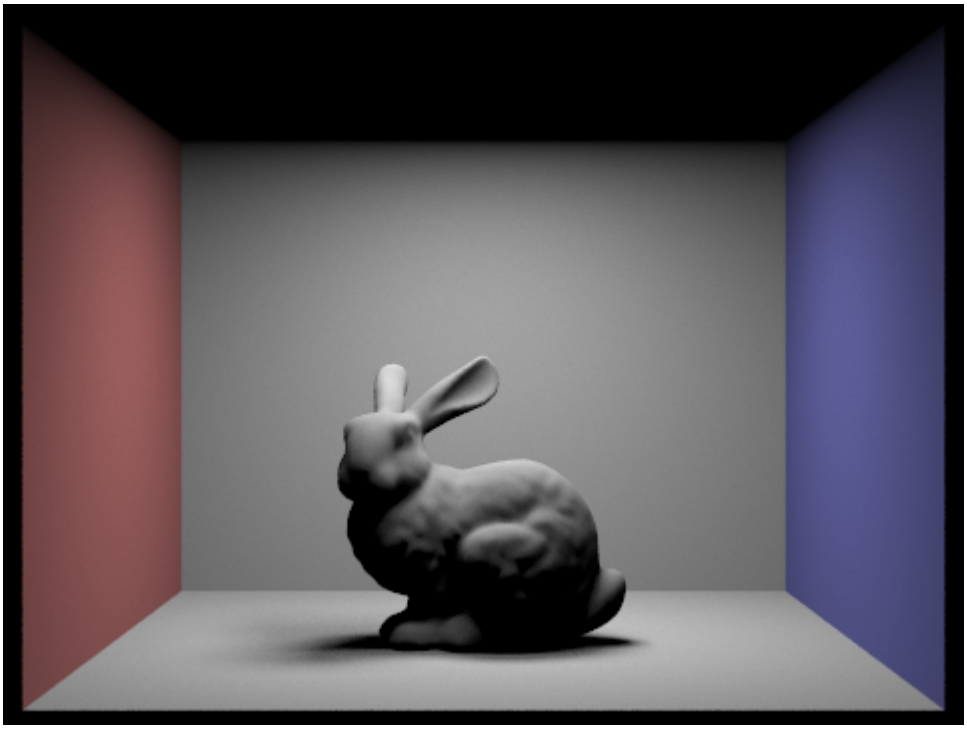
While this produces adequate images, the images are a bit grainy. This is in part because of taking too few samples and the randomness in finding the light.

As an alternative, we will implement importance sampling which will instead directly iterate over the lights in the scene instead of randomly sampling from the hemisphere which is normal to the intersection. The idea here is that if we know all the lights, we dont have to depend on random luck to find all the sources of light hitting the surface. Here are some of the improved renders which use the same number of samples as in hemisphere sampling.

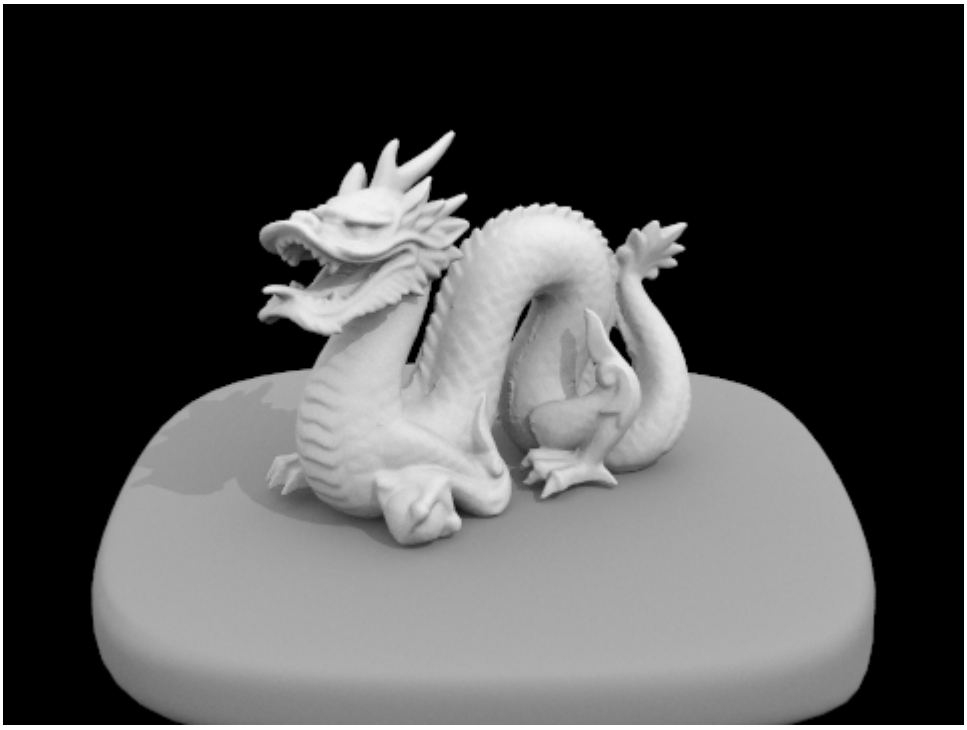
Important Spheres This Time



Important BunBun



Really Important Dragon



As you can see, even with the same number of samples we can get really smooth renders out when we use importance sampling.

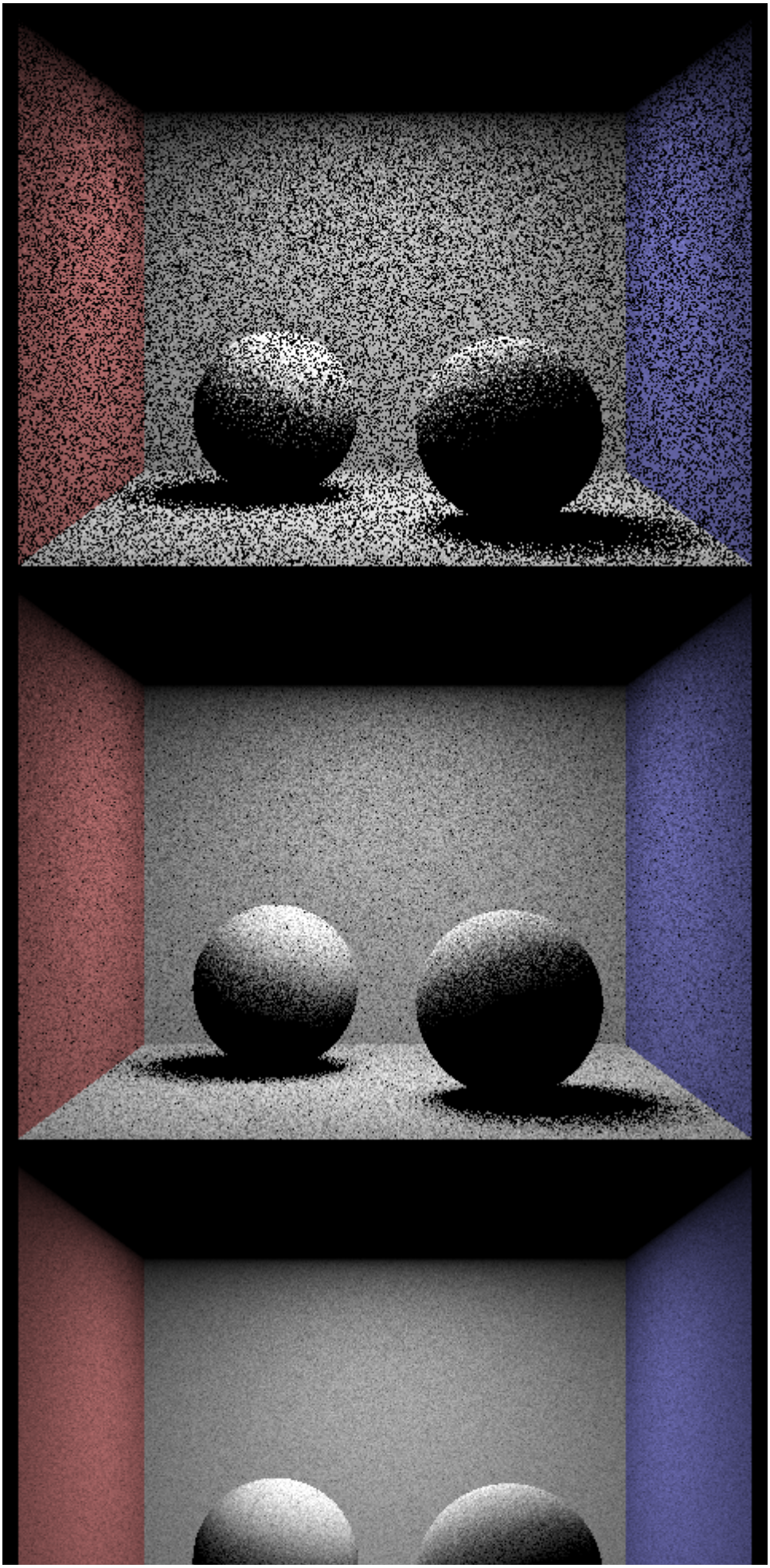
Analysis of Hemisphere vs Importance

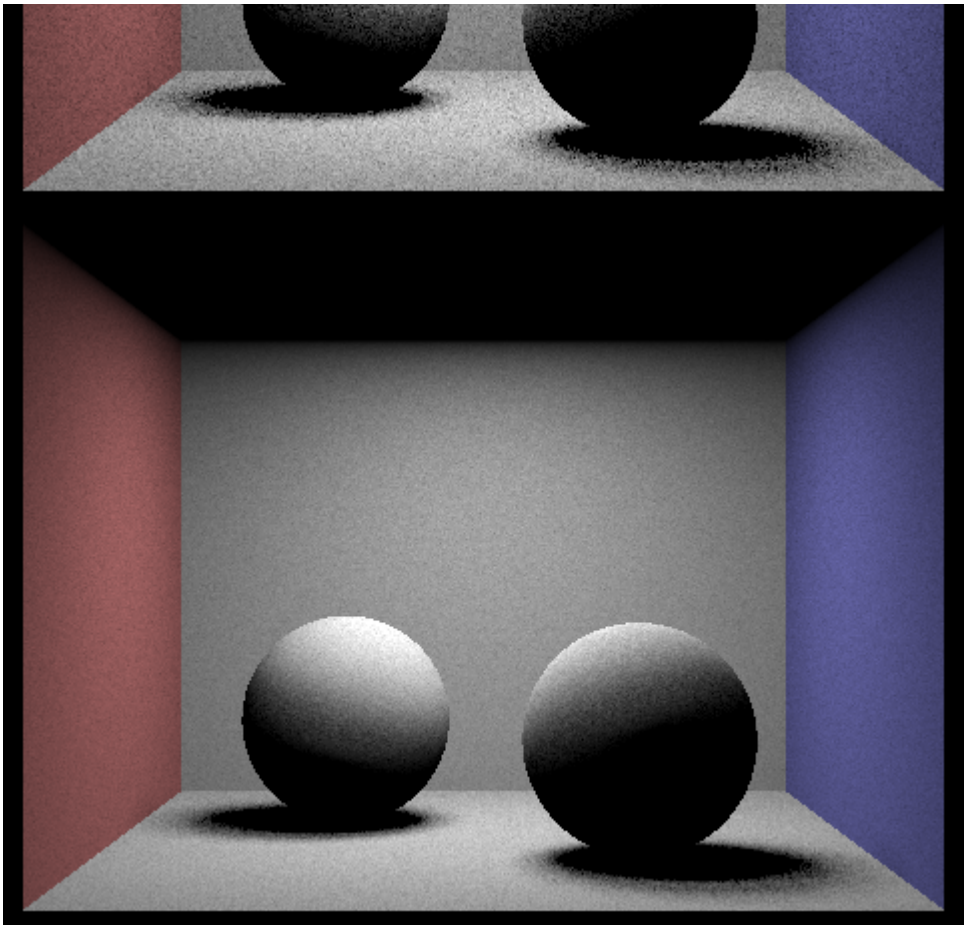
Hemisphere sampling is a simpler method and with enough samples produce decent images. The issue is that since there is randomness, it is very possible some intersection points do not collect light from the lights as much as it should. Importance sampling produces really smooth images because it guarantees we gather light from every light in the scene. There is not much of a differnece in runtime as far as I can tell so it seems it would be best to use importance sampling.

Light Ray Progression

One parameter that is relevant to importance sampling is the number of samples we take to each light in the scene. Since the light can span a large area, the light emitted can be from anywhere on that primitive and we randomly sample from the full area. Here is a progression 1, 4, 16, and then 64 samples done per area light.

Spheres in the Light





The area of interest are the shadows cast by the ball. These shadows are caused by the fact the light from the ceiling is blocked by the balls. This creates shadows which should get increasingly light as it gets farther away from the ball itself. When there is only one sample however, these shadows are either fully black for fully lit. As we increase the number of light rays, we get smoother transition in the shadows because we get a larger pool of rays to test and a portion of them do not hit the spheres. When we have 64 light rays, we can see there is a gradient in how heavy the shadows beneath the spheres are.

Part 4 - Global Illumination

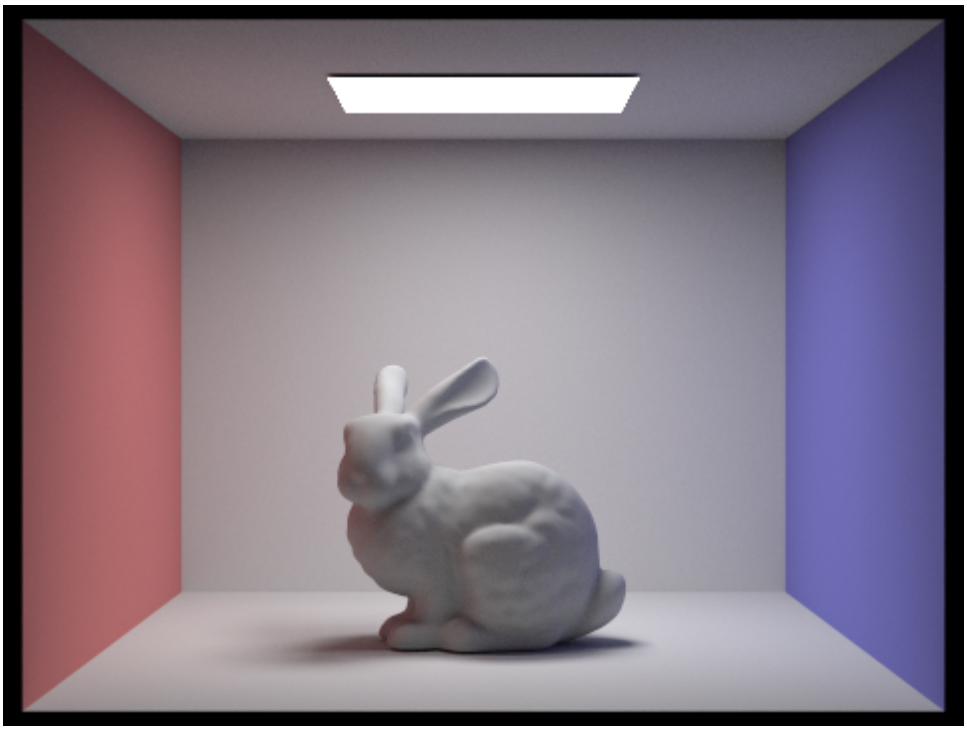
We have implemented the light gathered directly from the light bouncing off of the intersection point, but this is not an accurate representation of how light works. Rays can bounce multiple times off of surfaces and the light the camera receives from the intersection point is the combination of light from any number of bounces. We call the light which bounces more than once from the light to the camera indirect lighting.

To implement this, we now recursively calculate the light which reaches our intersection point. We first find the light that reaches this point directly from light sources using the techniques in part 3. Then we randomly pick a direction to sample based on the properties of the intersection point. Once we have a direction, we find an intersection in that direction and recurse on that new intersection point. We stop when we either hit a defined maximum ray depth or when the

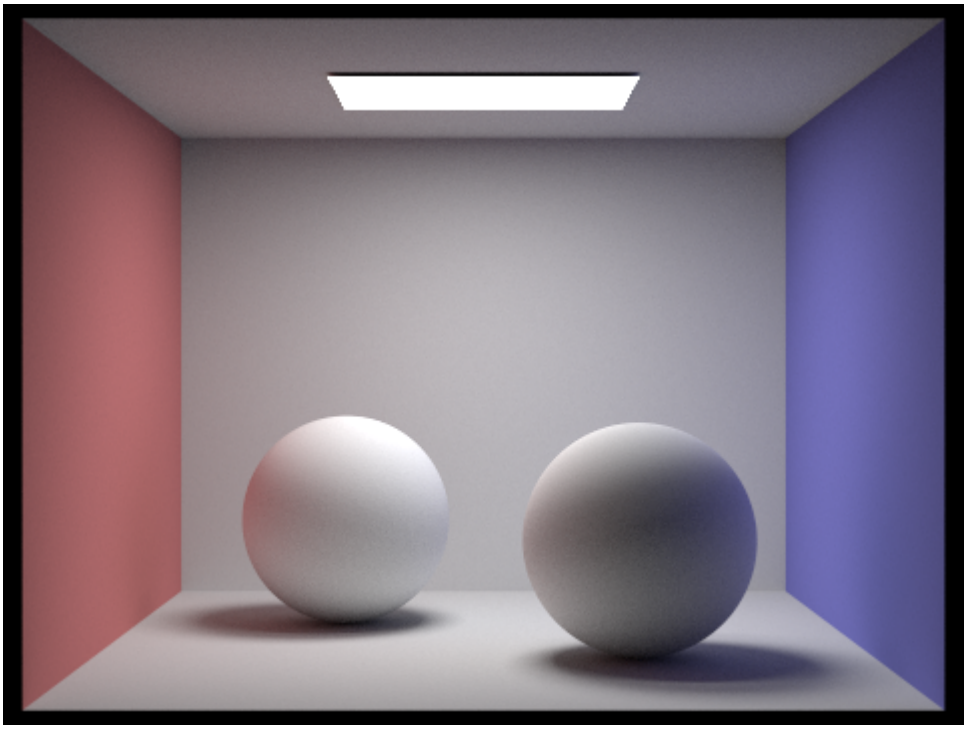
number of bounces has led to too much “light decay”. We can simulate this light decay by adding a probability of termination to our recursive light ray. This way, rays which bounce too many times are “likely” to stop which mimics the fact that after many bounces the energy of the light is mostly absorbed.

We add the resulting recursively calculated light from the new intersection point to our direct lighting with some weighting. We weight the recursive light by dividing by the probability of picking that direction and by the probability of continuing the light ray. This is sent back to the camera as the light from this intersection point. Here are some examples of global illumination.

Pretty Bunny

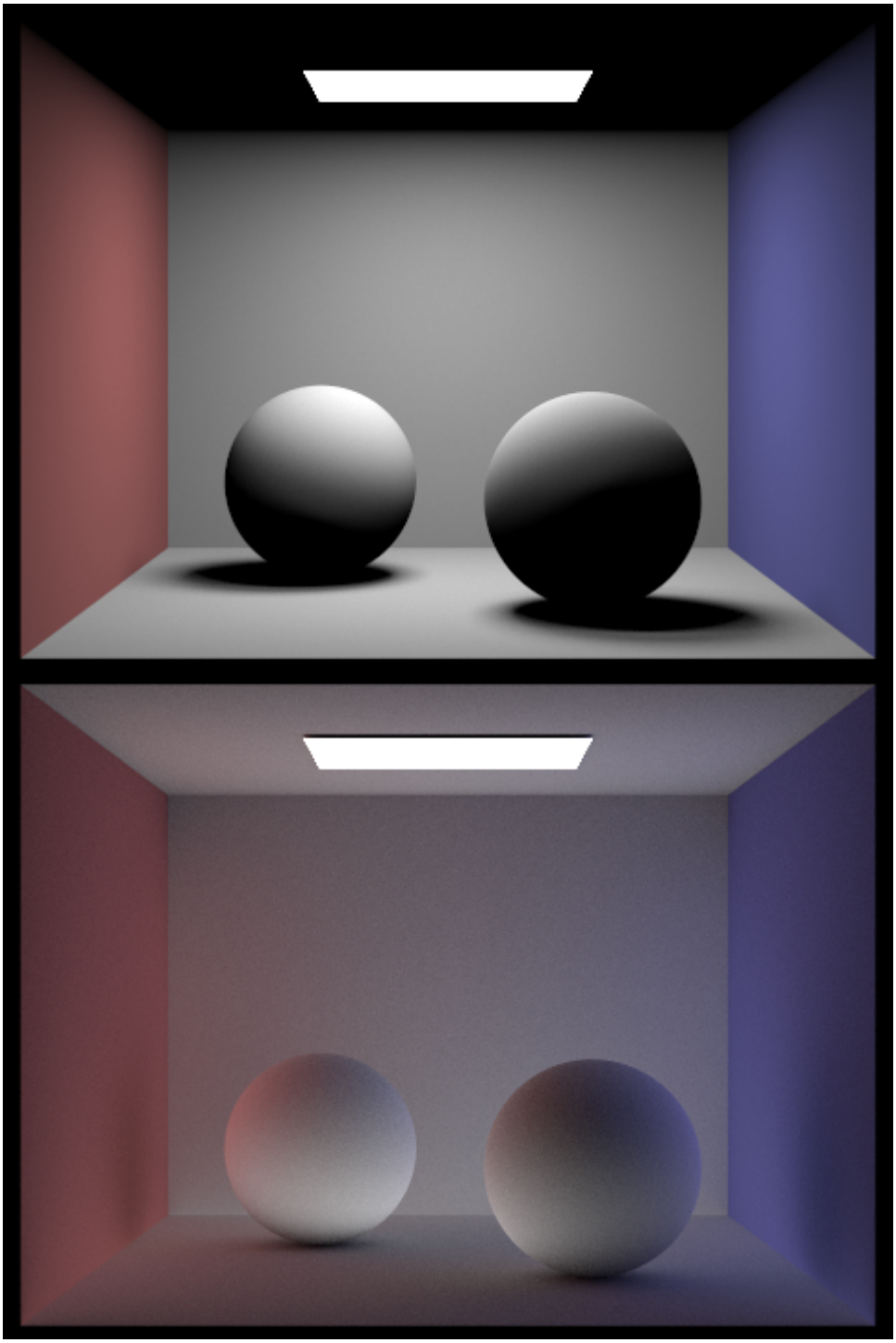


Smooth Spheres

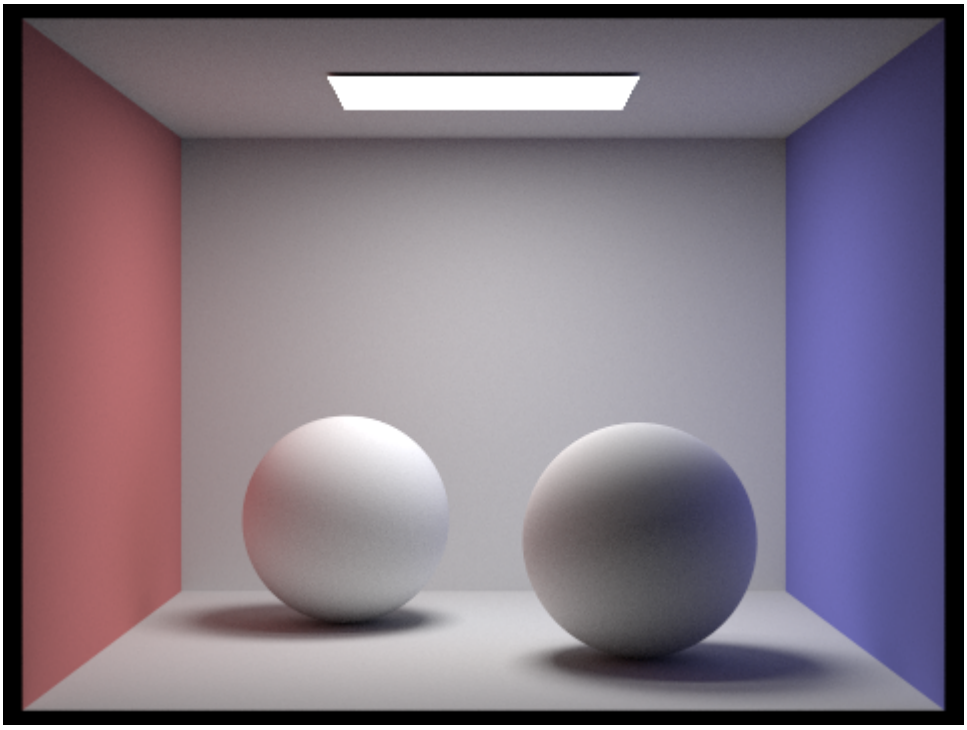


Direct vs Indirect Lighting

The improvement from global illumination is very noticable. We can break down global illumination to both direct and indirect lighting. Here are our trusty spheres again but with only direct illumination and then only indirect illumination.

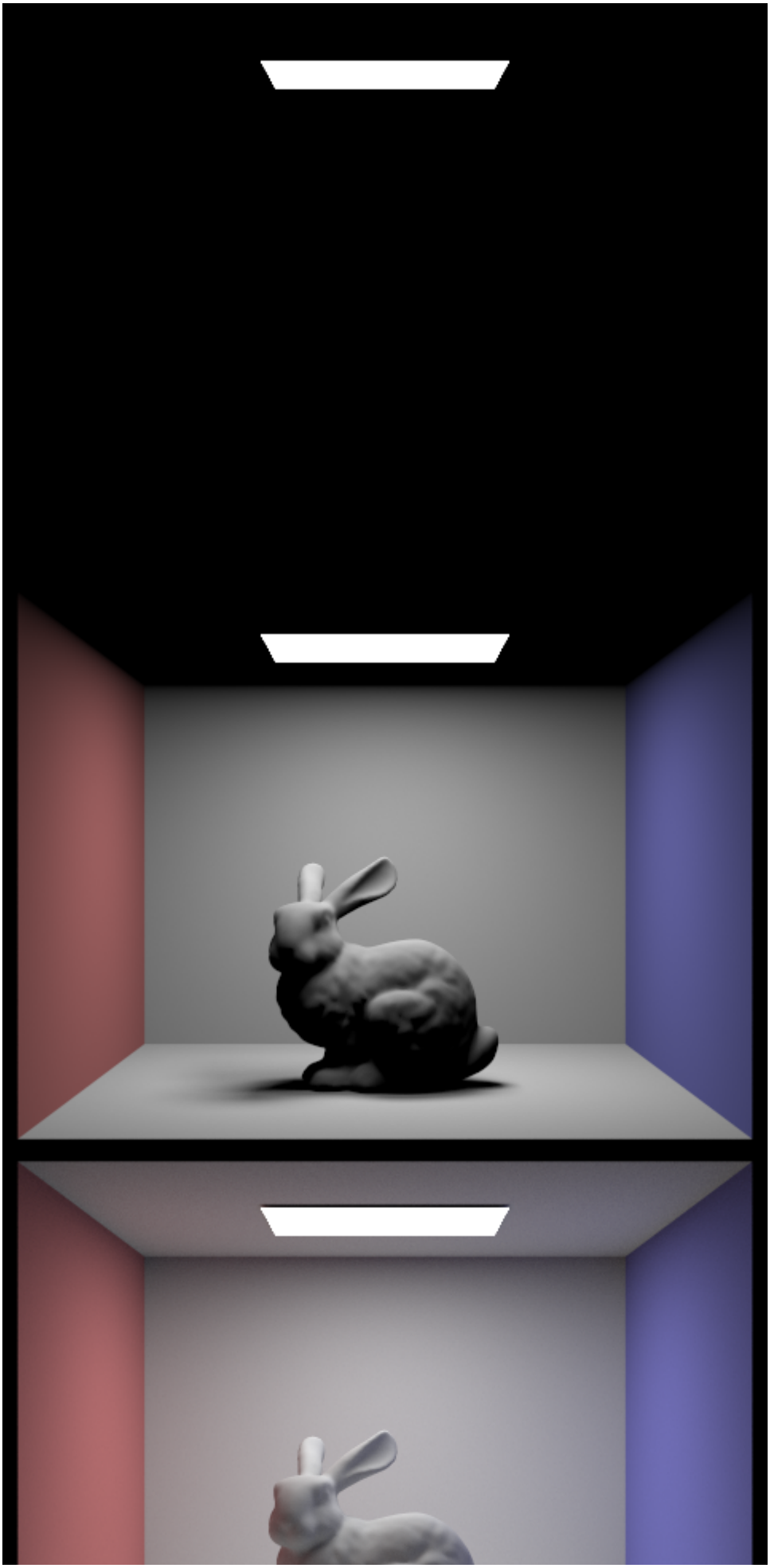


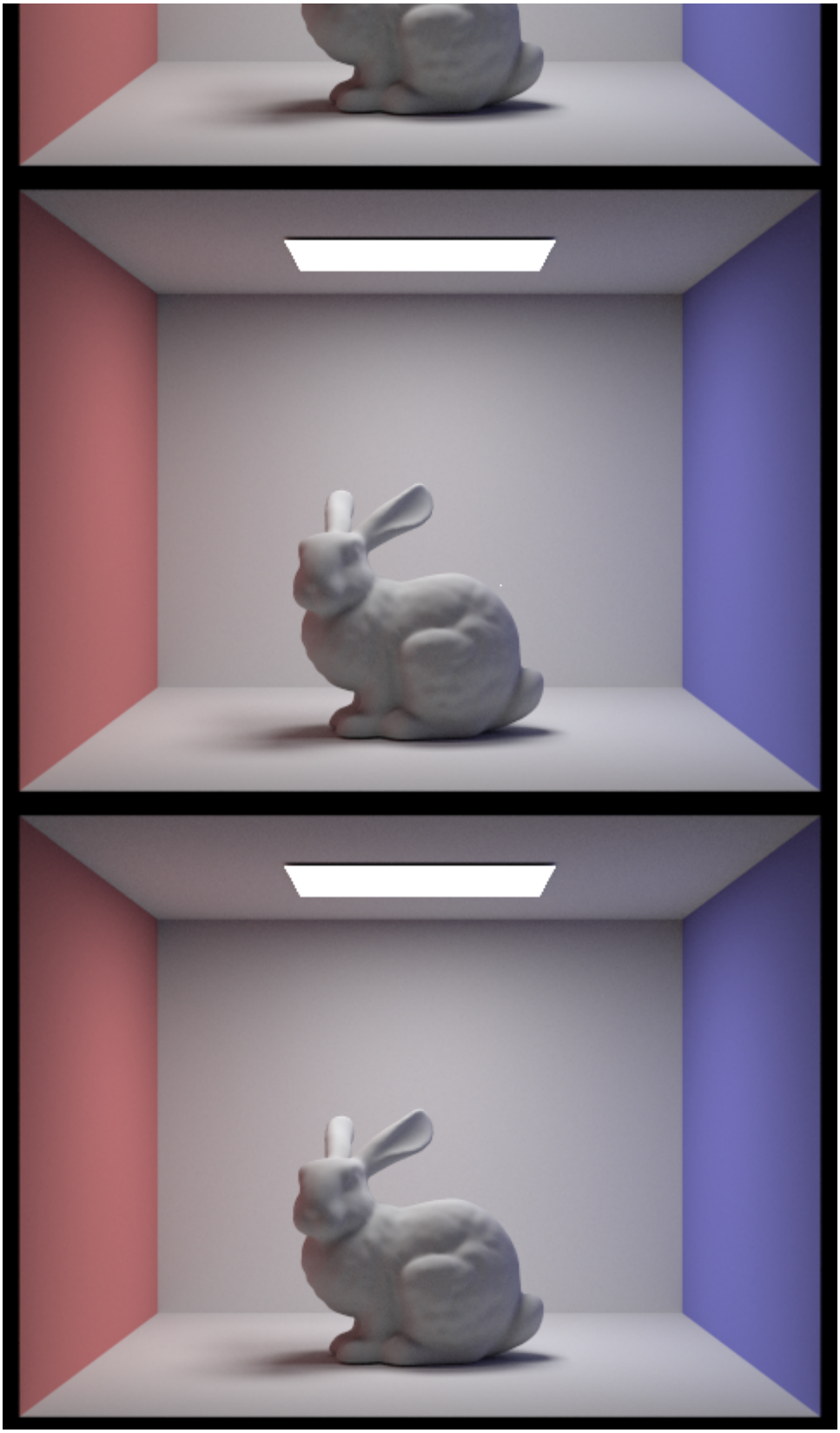
The direct illumination is the same as the result from part 3. The indirect illumination is pretty intersting as the brightest part of the spheres are not as bright. It also highlights the ambient light cast on to the side and bottom of the spheres because of the light bouncing off of the walls and the floor. If we summed up the two pictures, we'd get the picture below of full global illumination.



Max Depth Progression

We will take a look at the progression of images as we increase the max depth of the rays calculated for indirect illumination. Here are the images of the bunny with max ray depths of 0, 1, 2, 3, and then 100.



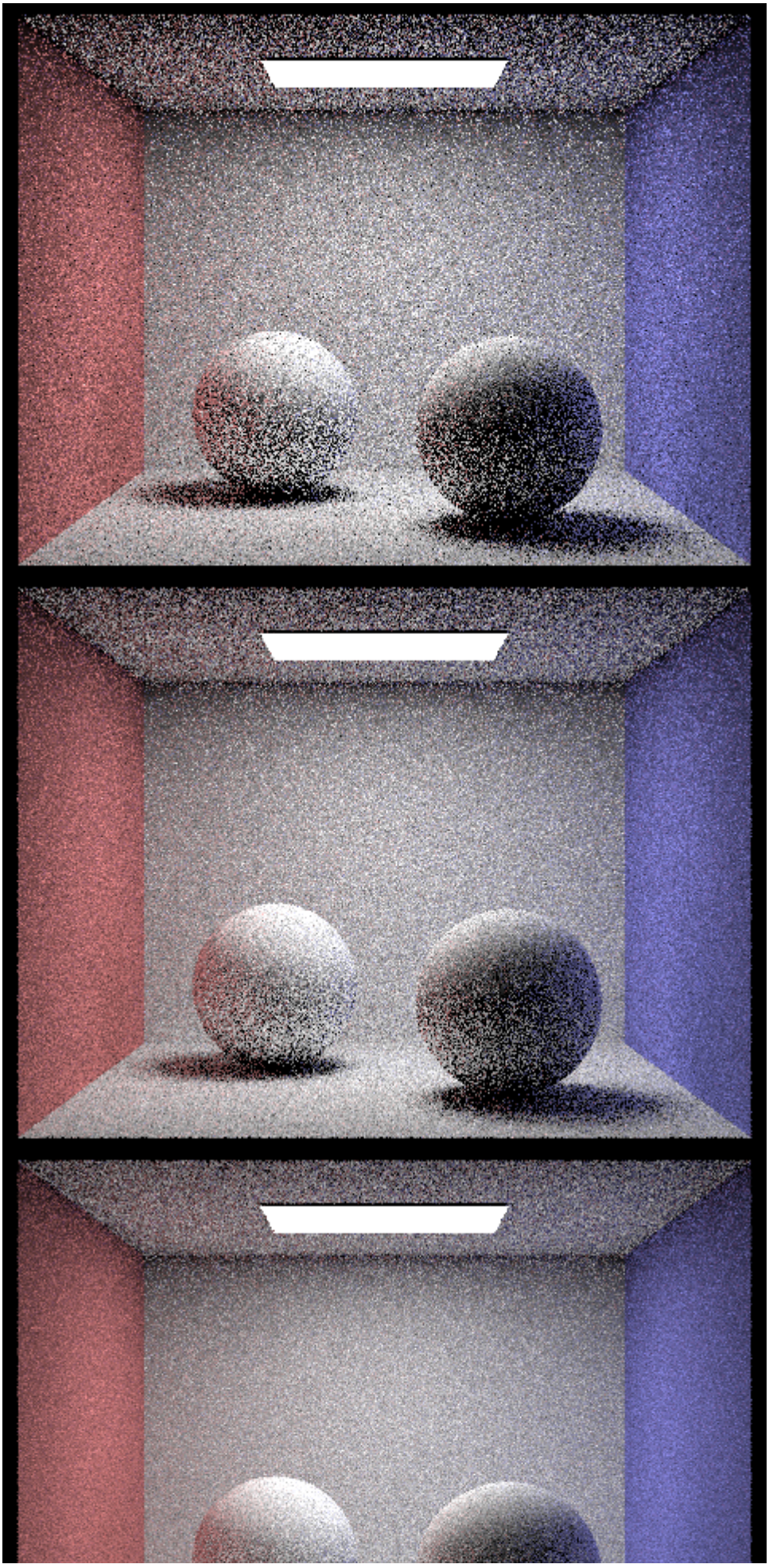


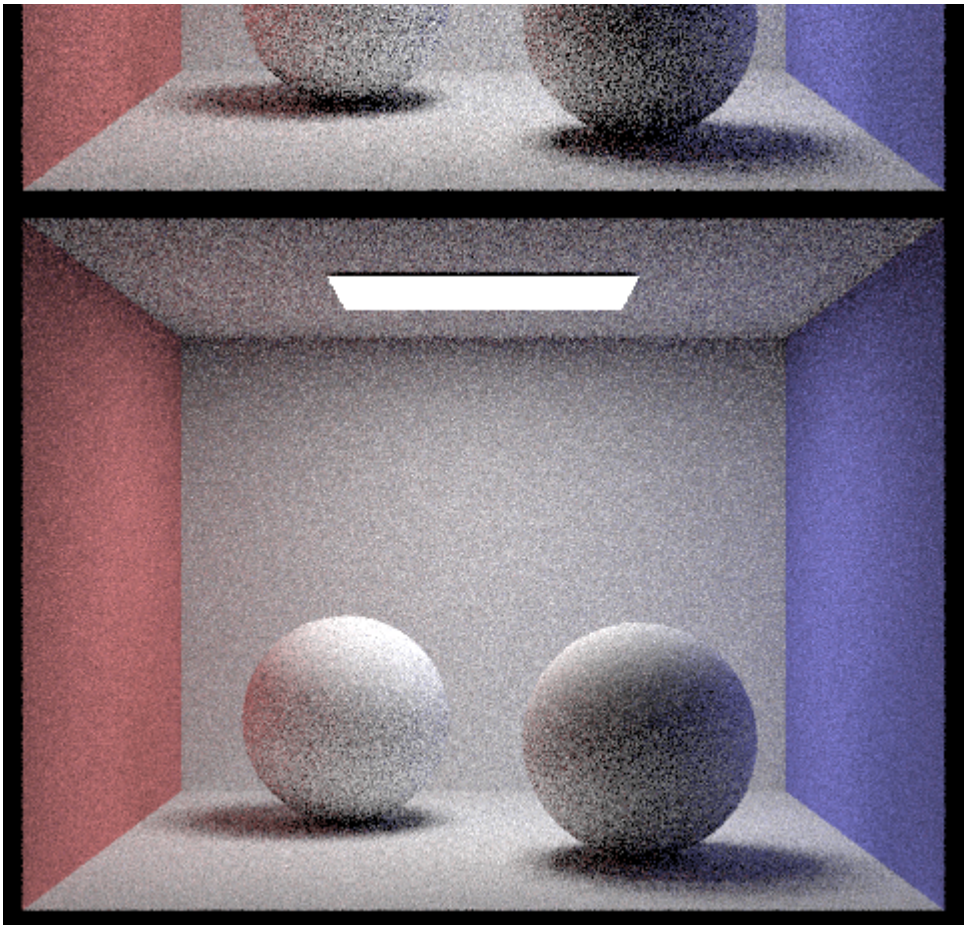
At depth 0, only things which emit light directly to the camera are captured so the scene is dark except the ceiling light. At depth 1, we have direct lighting which is the same as in part 3. With depths 2, 3, and 100 we now have indirect lighting which is significantly different from depth 1. The difference between 2 and 3 is very slight in that the corners of the room are slightly brighter. This makes sense as allowing more bounces means more light can be gathered. The difference between 3 and 100 is not noticable at all because of the termination probability.

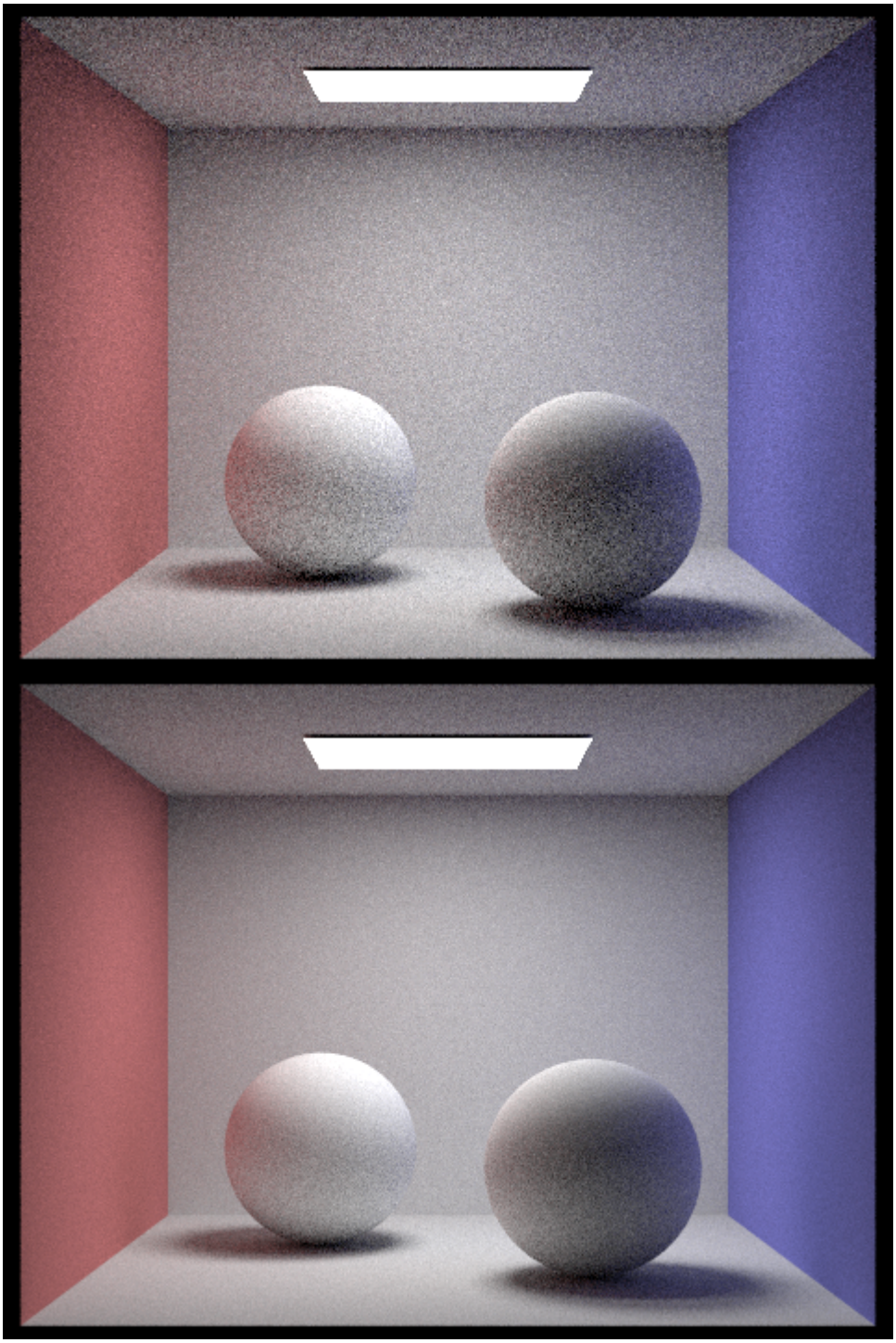
Since I used a probability of 0.7, it is pretty unlikely for a ray to continue past 3 bounces anyways and any which do have a very low weighting. This means a depth of 3 when the probability is 0.7 is probably pretty close to ideal.

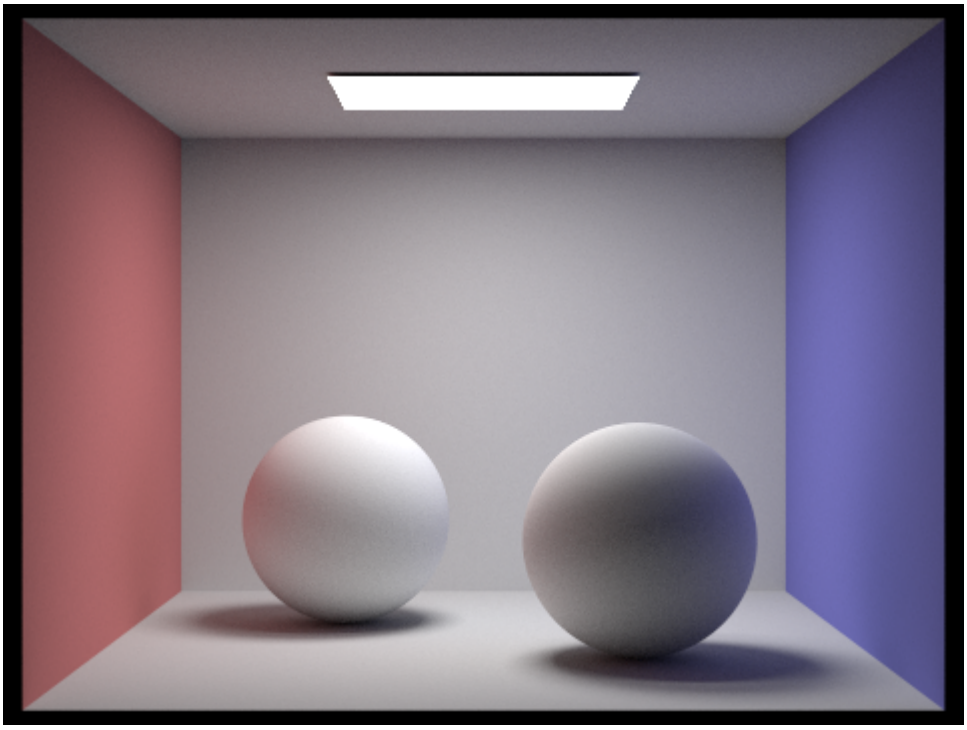
Sample Rate Progression

Another important parameter for illumination is the sample rate. Here is a progression of renders of the sphere with 1, 2, 4, 8, 16, 64, and then 1024 samples per pixel.









As we increase the samples, the amount of noise decreases significantly. The noise is mostly in the form of white specks which makes sense since if we only sample once, it is possible the bounce is coming from the white wall or ceiling. So with only 1 sample we get noise from the bounced light. With more samples, we average out the light which bounces from multiple directions. This gives smoother and more realistic renders because it is more representative of all the rays which are reaching this intersection point. At 64 samples the image looks like it has realistic lighting but it looks grainy because of the small noise. By 1024 samples the image looks really smooth and realistic.

Part 5 - Adaptive Sampling

One major issue of rendering is how long it takes to get enough samples such that the image looks clean. The high number of samples is needed for complex surfaces, in order to get realistic and smooth lighting, but is not really necessary for things like walls. To accommodate for this, we will implement adaptive sampling.

The idea here is that we will set a high maximum number of samples per pixel, but will provide a condition to stop early. The idea for the termination condition is that if after a bunch of samples and the running sample is mostly stationary, then more samples are unlikely to make a difference. We use a confidence interval determine how tightly we are narrowing down to the true value. If it is close, then we terminate early. With this we can also display a heat map showing how many samples were taken at each pixel. Blue is low sampling, red is high sampling, and green is in between. Here are our trusty slick spheres and a cute bunny with their adaptive sampling heat map.