Global Illumination: Monte Carlo Ray Tracing

MIT 6.837 Computer Graphics Final Project

Sophia Kwon

Motivation

Simple ray tracers that only utilize direct illumination, such as the one we implemented in assignment 4, have several limitations. Scenes rendered using this technique are usually fairly dark, as they fail to account for the indirect light that bounces around between the surfaces of the scene. In order to take this indirect light into account, we must utilize the rendering equation. The rendering equation finds the light at a certain point in the scene by taking the integral of incoming light over the hemisphere around the point. However, calculating the incoming light to our point from another point requires us to compute the integral over the hemisphere around that point as well, and we would continue to recurse. This makes the rendering equation impossible to solve analytically in most cases, so we must use an approximate solution. My approximation method of choice is Monte Carlo ray tracing, which calculates the incoming light at a certain point in the scene by sending rays out from that point in random directions and calculating the light at the points that those random rays hit the scene. These random rays approximate the integral over the hemisphere using Monte Carlo sampling.

Our assignment 4 ray tracer also had some other limitations. For example, the point light sources that we used in assignment 4 caused objects in the scene to have very hard, sharply defined shadows, which makes the image look somewhat unrealistic. The ray tracer was also limited in the types of materials it was capable of rendering: it could only recognize diffuse, shiny, and specular materials. To make the ray tracer more flexible, I decided to implement area lights (for softer shadows) as well as glossy and refractive materials.

Background work

I started off with my ray tracer from assignment 4. I based my work on the concepts from our global illumination lecture, as well as some lecture notes on Monte Carlo path tracing from UC Berkeley[1]. I also looked to a couple Monte Carlo Ray Tracing projects on Github[2] for more details.

[1] https://inst.eecs.berkeley.edu/~cs294-13/fa09/lectures/scribe-lecture5.pdf
[2] https://github.com/kbladin/Monte_Carlo_Ray_Tracer and https://github.com/JamesGriffin/Path-Tracer

Approach

The following subsections describe the features I added, in the order I added them, with descriptions of each implementation.

Monte Carlo Path Tracing with Antialiasing

I implemented Monte Carlo path tracing, with a recursion depth of 1 random ray per path. This allowed my ray tracer to send multiple rays from the eye through each pixel. Once each ray hit the scene, it sent out another ray in a random direction in order to get the light that would bounces from that random ray's hit point to the point we're trying to shade. Because the recursion depth was 1, these random rays did not send out more random rays from their hit points; each path consisted of one eye-to-pixel ray and one random ray.

The implementation of this feature works as follows. When calling my program, I can specify a number *n* of eye-to-pixel rays to send per pixel. For each pixel, it generates the ray from the eye and calls traceRay. We set the pixel color to be the average of the *n* returned traceRay values. Within a call to traceRay, random rays are created by randomly generating spherical coordinates theta angle between 0 and 2*pi and phi angle between 0 and pi/2, and rotating the normal of the surface from which we want the ray to originate by that theta and phi. This ensures that I am randomly sampling the hemisphere around a point. The ray tracer then calls the traceRay function on that random ray, but with an argument specifying that we do not wish to send more random rays from this one. The indirect component of illumination from this path is then calculated as the product of the material's diffuse color, the material's reflectance, and the return value of calling traceRay on the random ray.

I also implemented anti-aliasing in my Monte Carlo path tracer. I modified my generateRay function to add a random jitter to each ray so that each eye-to-pixel ray goes through a different random point on the pixel.

Area Lights

I added area lights as another light class. To specify an area light, the scene parser takes in 3 of the 4 points of the rectangular area light, a color value, and a flux value for light through the area light. The AreaLight object constructor takes in those values, calculates the 4th point's coordinates and the normal to the area light, and stores a mesh of the area light itself. It also calculates the area of the light and the radiance from the light, which is the flux times the color divided by area. The getIllumination function for AreaLight works by randomly selecting a point on the light, and calculating the vector and distance from point p to that point on the light. The intensity is calculated as the product of the light's radiance value, its area, and the cosine between the light's normal and the vector from the light to p, divided by the quantity (2 * pi * distToLight^2). That way, the illumination at some point from an area light is proportional to the inverse square of the distance between that point and the light. In order to render the actual area light in the scene, I checked for

Multiple Indirect Lighting Bounces

My next step was to allow my Monte Carlo ray tracer to send more random rays from the hit point of a random ray, i.e. to allow my indirect lighting to have a recursion depth of greater than 1. In my indirectComponent and traceRay functions, I added an argument indirectBounces to specify the maximum number of further random rays that this path could generate. When this parameter is 0, the path does not generate any more random rays and therefore does not recurse any further. When a call to indirectComponent sends a random ray, it then calls traceRay on this random ray, as before, but now with the indirectBounces parameter decremented by 1. I used Russian roulette termination to decide when to stop recursing. At any given point on a path, if the path is not at the max number of recursions yet, my ray tracer nondeterministically decides whether to cast another random ray or to end this path.

Additional Materials

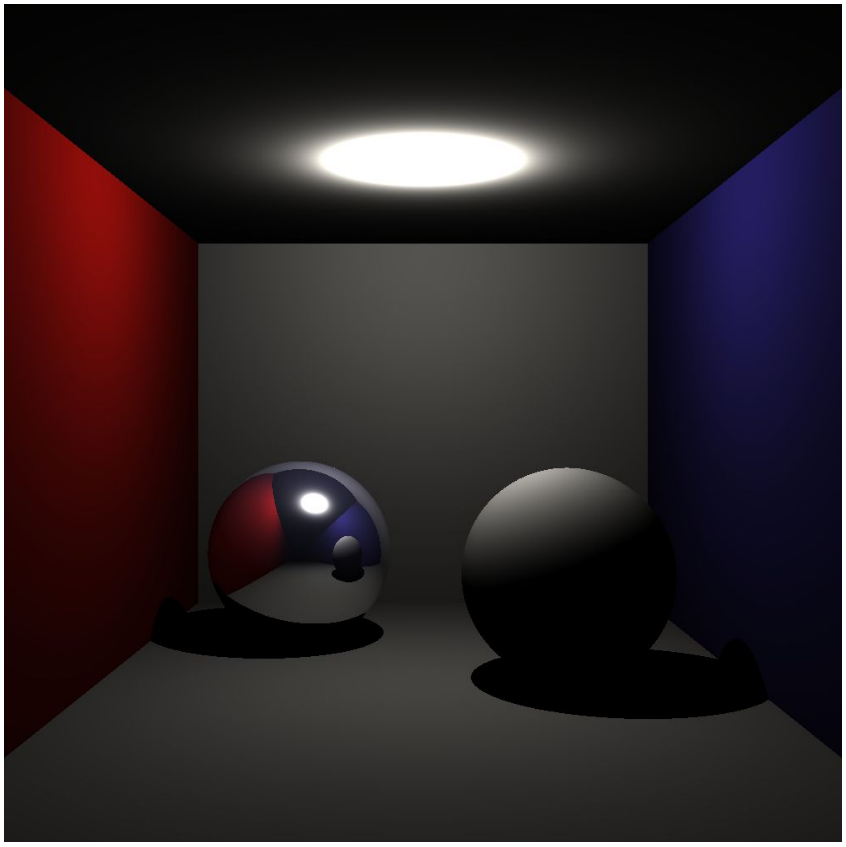
I implemented support for rendering glossy materials with my ray tracer. This works in the same way that rendering specular materials does, except that the perfect reflection ray (as used in specular materials) is jittered by some small random amount. When specifying the material, the input also specifies a glossiness value, which is the maximum angle over which the reflection ray can be jittered. The closer the glossiness value is to 0, the more similar the glossy material will look to a perfectly specular material. The jitter is implemented by selecting a random theta between 0 and 2*pi and a random phi between 0 and half the glossiness value, and rotating the perfect reflection vector by theta and phi.

I also support refractive materials in my ray tracer. When shading a refractive material, the ray tracer does not calculate the direct lighting, indirect lighting, or specular components of the illumination. Instead, it calculates the refraction component of illumination. It first finds the refracted ray using the formula from the ray tracing lecture:

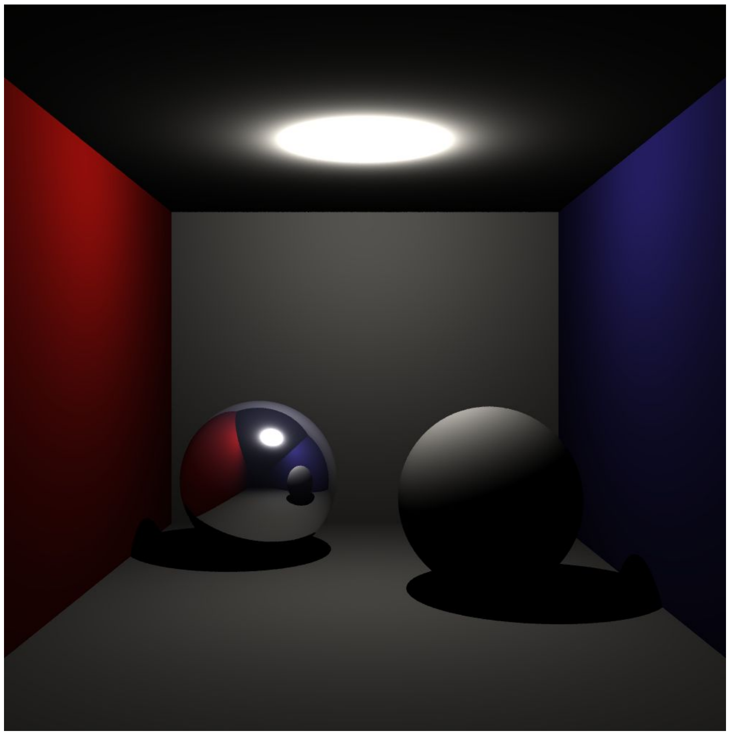
$[n_r(N \cdot I) - \sqrt{1 - n_r^2\,(1-(N \cdot I)^2)}\,]\,N - n_r I$. I hard coded n_i and n_T to be the indices of refraction of air and glass (order depending on whether the ray was inside the object coming out or outside the object going in). To check whether the ray was coming from inside or outside the object, I compute the dot product of the surface normal and the ray's direction. If the dot product is positive, the ray is coming from the inside of the object going out, so n_r is the ratio of the indices of refraction of glass to air instead of the other way around, and we flip the surface normal. The refraction component of illumination that we return then is the material's diffuse color times traceRay called on the refracted ray.

Results

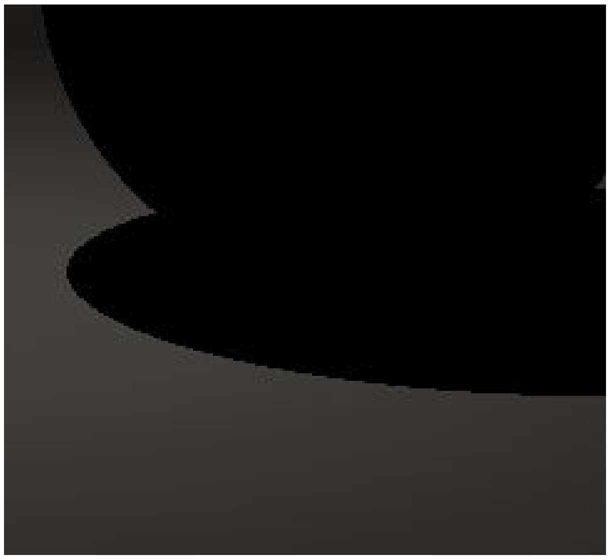
The following subsections describe the results of adding each feature, with images of the rendered scenes. For reference, here is a scene rendered using my original ray tracer from assignment 4.

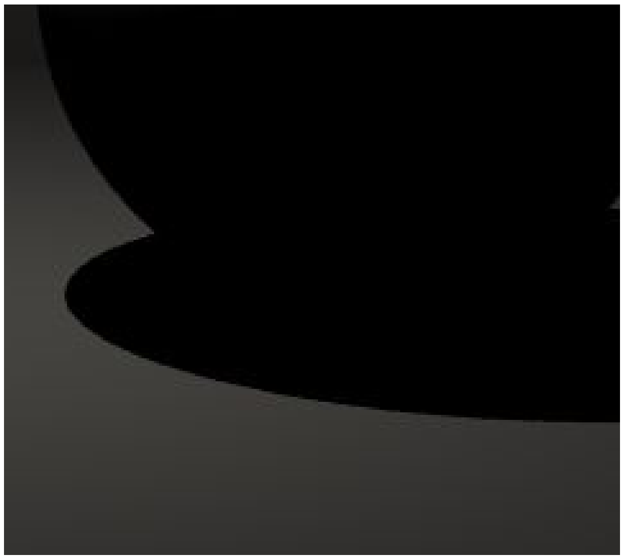


Monte Carlo Path Tracing with Antialiasing

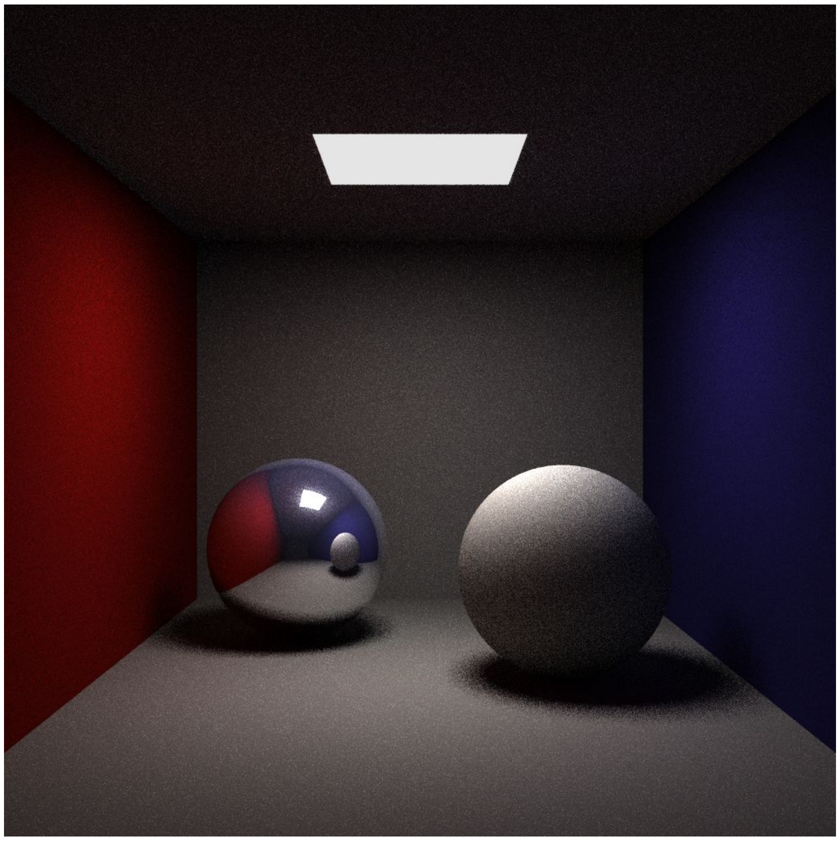


Above is the same scene rendered using Monte Carlo ray tracing with antialiasing. The scenes rendered using Monte Carlo path tracing with antialiasing look pretty similar to the ones rendered using the original ray tracer. One noticeable difference is that the original ray tracer produces some jagged edges, which the Monte Carlo path tracer smooths out using antialiasing. Below, the left image is a zoomed in version of the scene rendered using the original ray tracer, and the right image is the same thing but with the Monte Carlo ray tracer, clearly illustrating the smoothing of the jagged edges in the shadow.

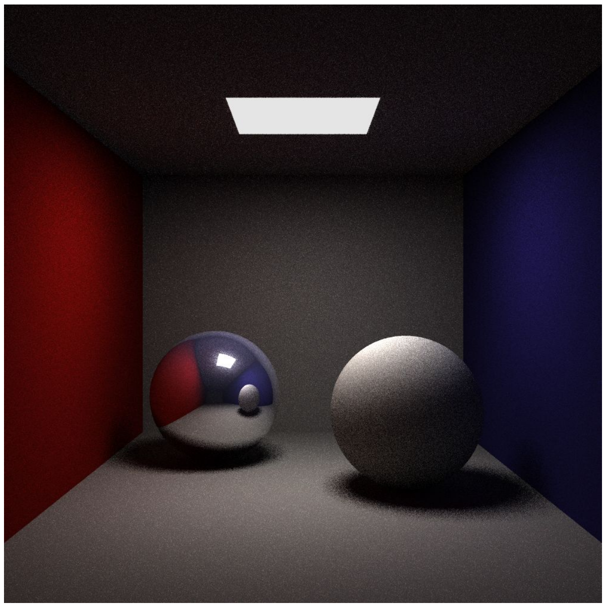
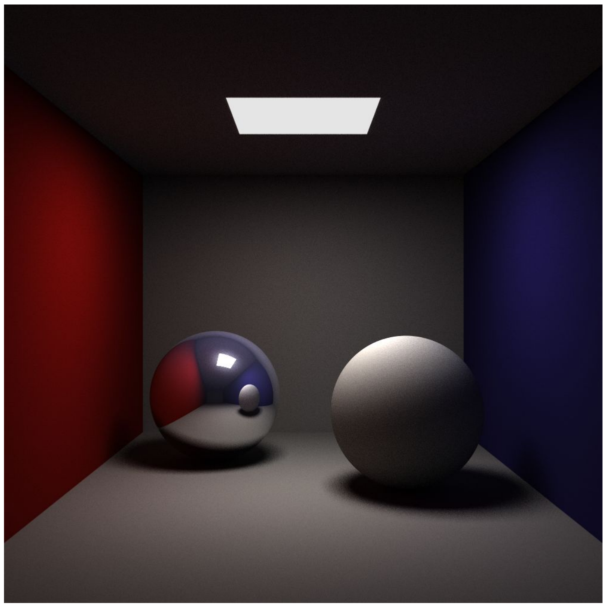




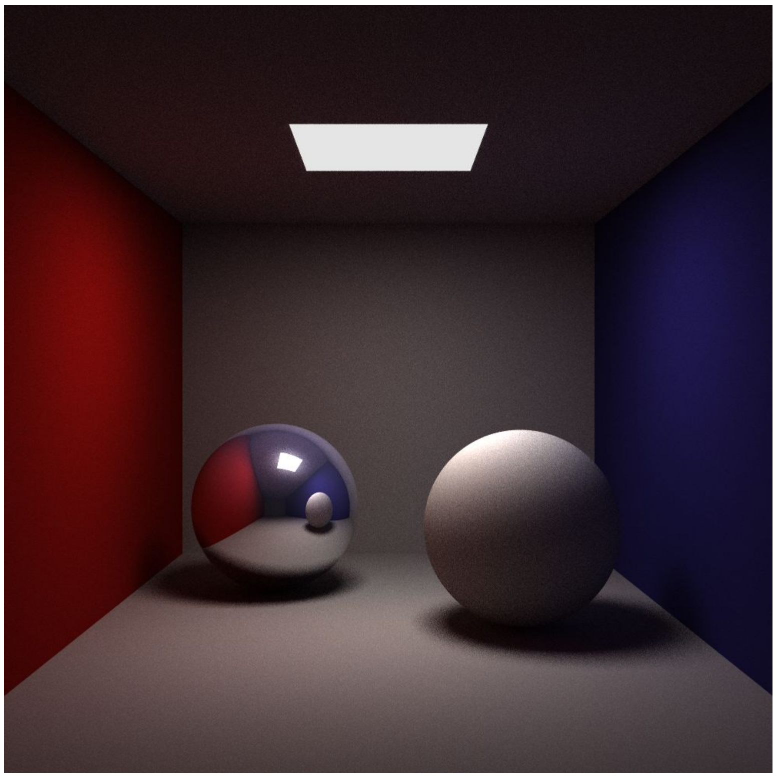
Area Lights



With area lights, the image has soft shadows, which looks much more natural than the hard shadows from point lights. One downside is that there is more noise in the image than before. However, the noise can easily be reduced by simply casting more rays through each pixel. In the images below, the left image was rendered using 10 paths per pixel, and the right image was rendered using 100 paths per pixel. The right image has much fewer noisy pixels than the left image does.

Multiple Indirect Lighting Bounces

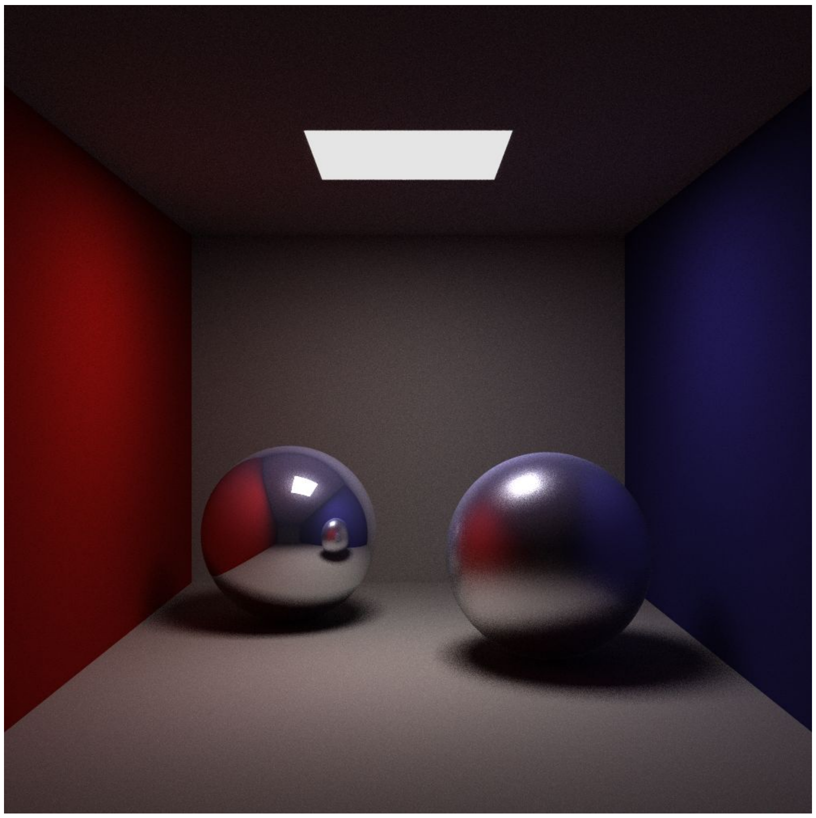


This scene was rendered using a max recursion depth of 10, which meant that any path could bounces around the scene at most 10 times before the path terminated. This image is overall a little brighter than the previous images, since the longer paths mean that more light bounces around the scene. This image also has visible color bleeding, especially on the ceiling.

This is a desirable effect because it means the light bounced off the colored walls and onto other surfaces, such as the ceiling. The Russian roulette termination did not significantly affect the quality of the rendered images, but it did significantly reduce the time required to render the images, which is great since any optimizations are welcome when it comes to ray tracing.

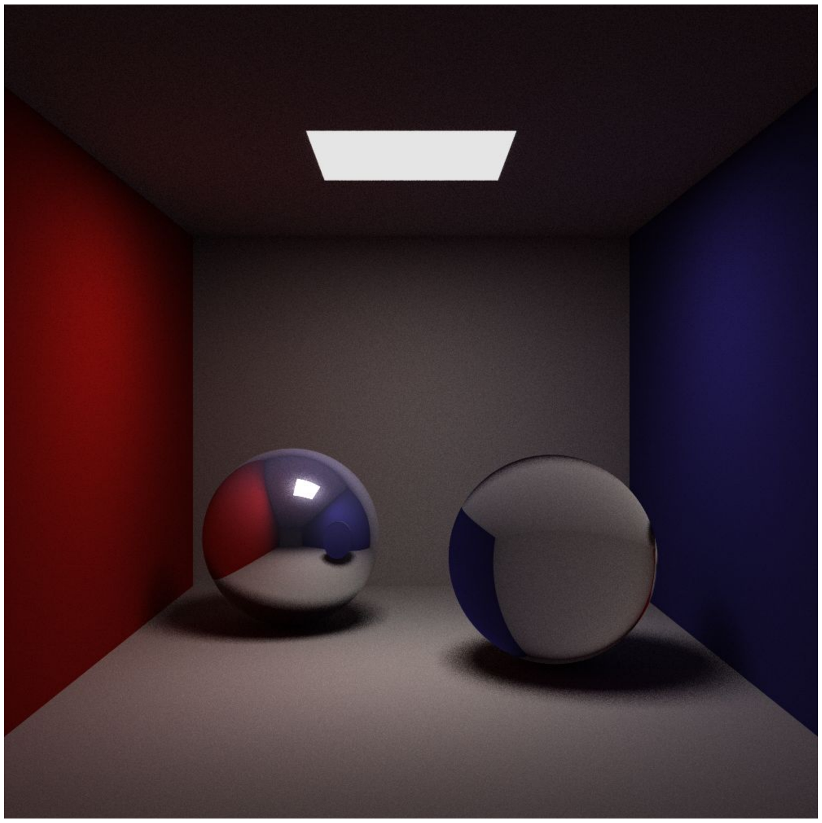
Additional Materials

Below is the same scene but with a glossy sphere on the right, instead of the diffuse sphere.



The right sphere clearly reflects light just as the perfectly specular sphere does, but the reflections are blurry because of the jittered reflections. By varying the glossiness value of the material, I could render a range of glossy materials.

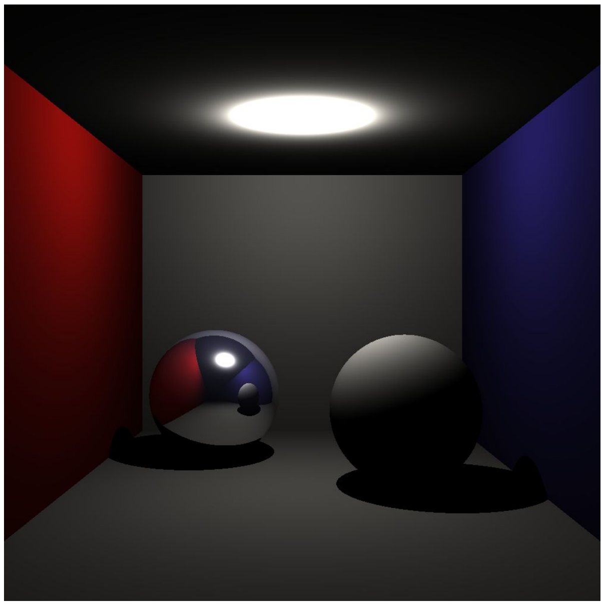
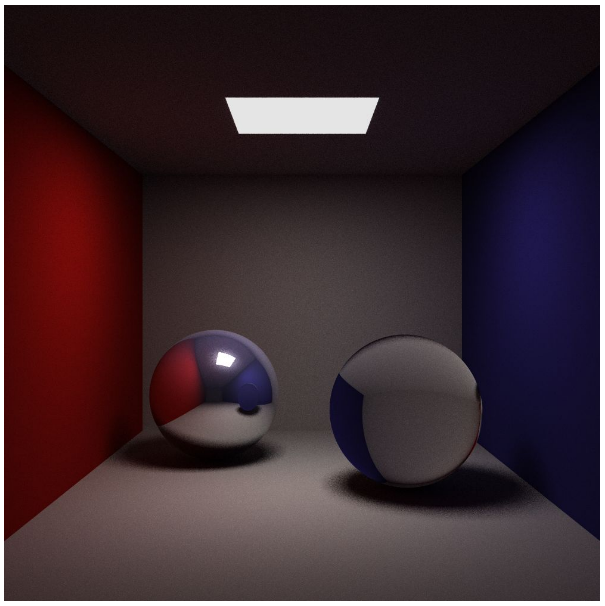
Below is the image again, but with a refractive sphere on the right.



The refractive sphere magnifies the area behind it because of the bending of light inside the sphere. There is no caustic underneath the sphere because my ray tracer does not include a photon map.

Conclusion

Below is an image rendered using my original simple ray tracer, and another rendered using my Monte Carlo ray tracer, with antialiasing, area lights, multiple indirect light bounces, and an additional material.

The features I implemented allow my ray tracer to produce more realistic images, and also gives my ray tracer a wider variety of scenes it is capable of rendering. A potential next step to extend this project would be to implement photon mapping, so that I could render caustics for refractive materials.