Ray Tracing in Python

A. Desai[†]

[†] *Dept. of Physics & Astronomy, Stony Brook University*

December 8, 2021

1 Introduction

It is amazing that so much of physics is essentially the study of light. While the theories have gotten more complicated as physicists have created better models for how the universe works, at its core much of it started by understanding something centered in the human experience: light. In this project I wanted to explore how interactions of light can be modeled through a concept called ray tracing. Ray tracing essentially is tracking rays of light and using physics to determine how they interact with objects of varying shapes and size. This can become enormously complicated as objects become more complex, so there is significant interest in developing ray-tracing algorithms, mostly due to the wide applicability of this technology to visual media. This problem is usually attacked by creating object classes, a light ray classes, and different physical models (equations) depending on the material of the object. Modeling every possible light ray from a given source or sources is generally unfeasible given the complexity and randomness of interactions, therefore Monte Carlo methods are usually used to get an approximate, yet accurate representation of how light would interact with various objects. These images are rendered and outputted as image files. I implemented my own ray tracer by following a series of books written by Peter Shirley that cover a variety of ray tracing methods [1,2,3].

2 Computational Approach

2.1 The basics

Like any complex program, this ray tracer was slowly built from simple concepts. The first important concept is the concept of what a 'ray' actually should be represented as. In this program, I represented it as a vector, with a direction and an origin. The difference here is that the ray needs to be flexible, unlike traditional vectors I want to be able to extend it along its axis to an arbitrary range in order to see what it interacts with. This is done by using a parameter 't' which represents how far along the path of the ray an event occurs.

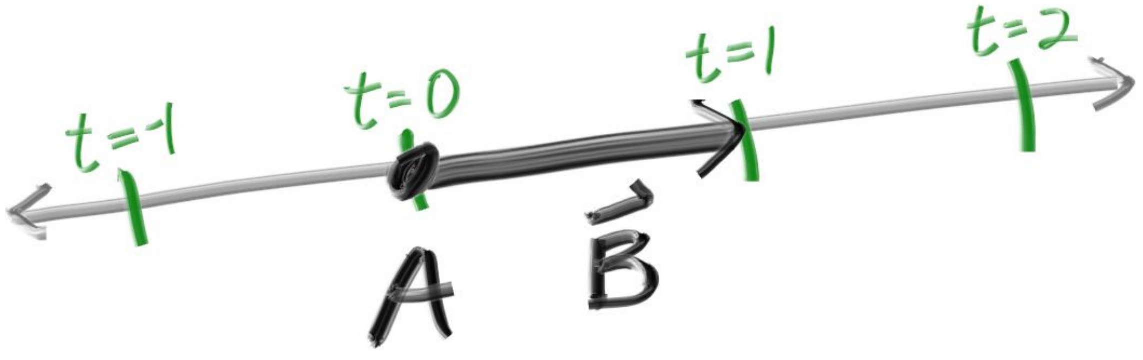


Figure 1: A ray [1]

This is nicely wrapped up in a ray class. The next question (and perhaps this should have been first) is how the image should be output. I chose the simplest method: a ppm file, which can be output in readable ascii format. The details of the file format are unimportant, the main idea here is it stores rgb values. This means when the ray interacts with its environment it returns a color vector. These images are rendered by creating two loops that loop through a grid of given height and width. Now it is easy to set up a background environment; just have your ray tracer return that background color as it iterates through the height (i) and width (j) of the image. The ray returning a color is coded using a function called ray_color. You can think of each (i,j) value as a pixel.

2.1.1 Antialiasing

Antialiasing describes a feature where instead of sending a single ray at each pixel, you send a specified amount of rays (in the code this parameter is the number of samples per pixel), and then averages all the returning color values. This allows for greater accuracy in images, though it significantly increases runtime.

2.2 Objects

The next step is to actually create objects for the ray to interact with, and one of the easiest things to calculate is how a ray interacts with a sphere. This calculation is fairly simple. If (C_x, C_y, C_z) label the points of the center of the sphere, then the equation of the sphere is $(x-C_x)^2+(y-C_y)^2+(z-C_z)^2 = r^2$. Here the vector from the center is $\mathbf{C} = (C_x, C_y, C_z)$. The vector from the center to point P is then $\mathbf{P}-\mathbf{C}$. Note now that $(\mathbf{P}-\mathbf{C})\cdot(\mathbf{P}-\mathbf{C}) = (x-C_x)^2+(y-C_y)^2+(z-C_z)^2)$ Then the equation of the sphere can be rewritten as $(\mathbf{P}-\mathbf{C})\cdot(\mathbf{P}-\mathbf{C}) = r^2$. This point can be varied with a parameter (as before when I discussed how to extend the ray) and let that parameter be called t, such that $\mathbf{P} = \mathbf{A}+t*\mathbf{B}$. So now the equation becomes: $(\mathbf{A}+t\mathbf{B}-\mathbf{C})\cdot(\mathbf{A}+t\mathbf{B}-\mathbf{C}) = r^2$. Here the unknown is t, and can easily be solved using the quadratic formula. Here no solution means the ray misses the sphere, one solution it means it touches the edge. Now there remains one more important feature before a tracer can actually give the impression of creating a 3-dimensional object, and that is the fact that these objects must have surface normals. The surface normal of a sphere at a given point $\mathbf{P}$ is just given by the vector $\mathbf{P}-\mathbf{C}$. These surface normals can be visualized with a color gradient (see Figure 2). In principle this same mathematical approach can be used to try and create any object, but in actuality it gets complicated very quickly. I only implemented one more object in this project, and those were axis aligned rectangles, which are relatively straightforward to compute. To determine whether a ray hits an axis aligned rectangle, you just have to determine whether it hits the corresponding plane, and if it does, you can check to see if it is within the rectangle boundaries. These rectangles can also be used to create boxes, and in a similar vein one can generally create complicated objects by using different combinations of simpler ones. In my program I defined my convention as having normals of objects point outwards. For axis aligned rectangles, I just chose the positive side to count as 'outside'.

2.2.1 Instances

While spheres have rotational and translational symmetry,rectangles and boxes do not. An easy way to handle when one actually wants to rotate or translate them is to create an instance. An instance is a faux object that just takes an oncoming ray, rotates/translates the ray, and then checks to see if the object was hit with this moved ray. This allows for general rotational and translational flexibility without needing to account for it in object geometry.

2.3 Materials

2.3.1 Diffuse Objects

Now that the basic geometry of objects can be stored and computed, the next obvious step is to give these objects a material, so we can study how light approximately interacts with them. Most everyday objects are diffuse, which can be described by a simple model. Light rays randomly scatter when they hit diffuse objects, and the diffuse object modulates these scattered rays with its own intrinsic color. They might also absorb oncoming light rays, and thus darker surfaces appear dark. There are many ways to approximate light randomly scattering off a surface, but the method

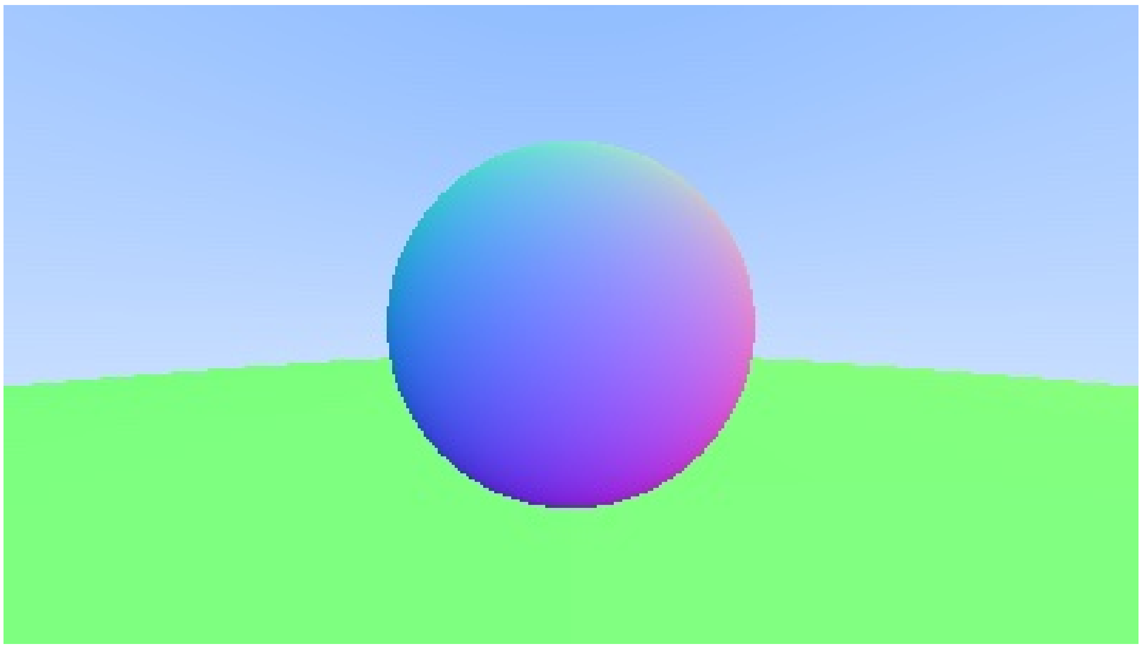


Figure 2: Here a very large green sphere is used as the ground, the background color defaults to a sky-like color, and the sphere in the middle has its surface normals mapped as a gradient.

I used here involves using a unit sphere. Take the point **P** where a ray hits the sphere, and call the normal of the sphere at this point **n**. If you then take the point $\mathbf{P} + \mathbf{n}$ and create a unit sphere around it, you choose a random scatter direction by choosing a point in that sphere.You can then make that point a unit vector beginning from the point P on the sphere, and now the direction of that vector is the direction that the ray scatters. You can modulate this light ray with the object's color just by keeping track of the color vector in ray_color. Note here that the ray scatters so you have to recur ray_color to follow along its new trajectory. Ideally you would want to follow the ray until it doesn't hit anything, but as this can take a long time there is a max depth parameter included so that after reaching this depth the ray stops returning any light. Rendering spheres with this new material, gives an image as shown in Figure 3. The material of a diffuse object is sometimes called a lambertian material.

2.3.2 Metal Objects

Metal objects are also interesting to model. For metal objects, light ray don't scatter randomly, they reflect. This can pretty easily be implemented, rather than have a ray choose a random direction to scatter, you use the angle of it hitting the normal vector to compute its final direction. Some metals don't reflect perfectly, but this can also be modeled with a fuzziness parameter. With a fuzziness parameter, the reflected ray shifts its direction slightly by once again choosing a point within a sphere, although this time the size of the sphere is dictated by the fuzziness parameter. To simplify this calculation somewhat, the fuzziness parameter is allowed only to range from 0-1, which allows for a unit sphere to be used when computing the random direction. For examples see Figure 4.

2.3.3 Dielectric Objects

For dielectric objects, rays are theoretically reflected and refracted. This interaction is governed by Snell's Law: $\eta * \sin\theta = \eta' * \sin\theta'$ The way my code handles dielectric objects is by first calculating whether an incoming light ray is refracted or not by looking at angle of the oncoming ray. If Snell's Law does not have a solution for this angle, the ray must be reflected.If Snell's Law does have a solution, then I randomly choose whether the ray is reflected or refracted, to avoid having to check branching rays. For real glass spheres, reflectivity r varies with angles. This is roughly approximated by a polynomial developed by Christophe Schlick. Let $r_0 = (\frac{1-\frac{\eta}{\eta'}}{1+\frac{\eta}{\eta'}})^2$ Then $r = r_0 + (1 - r_0) * (1 - \cos\theta)^5$ An example of a glass sphere can be found in Figure 5 and 9.

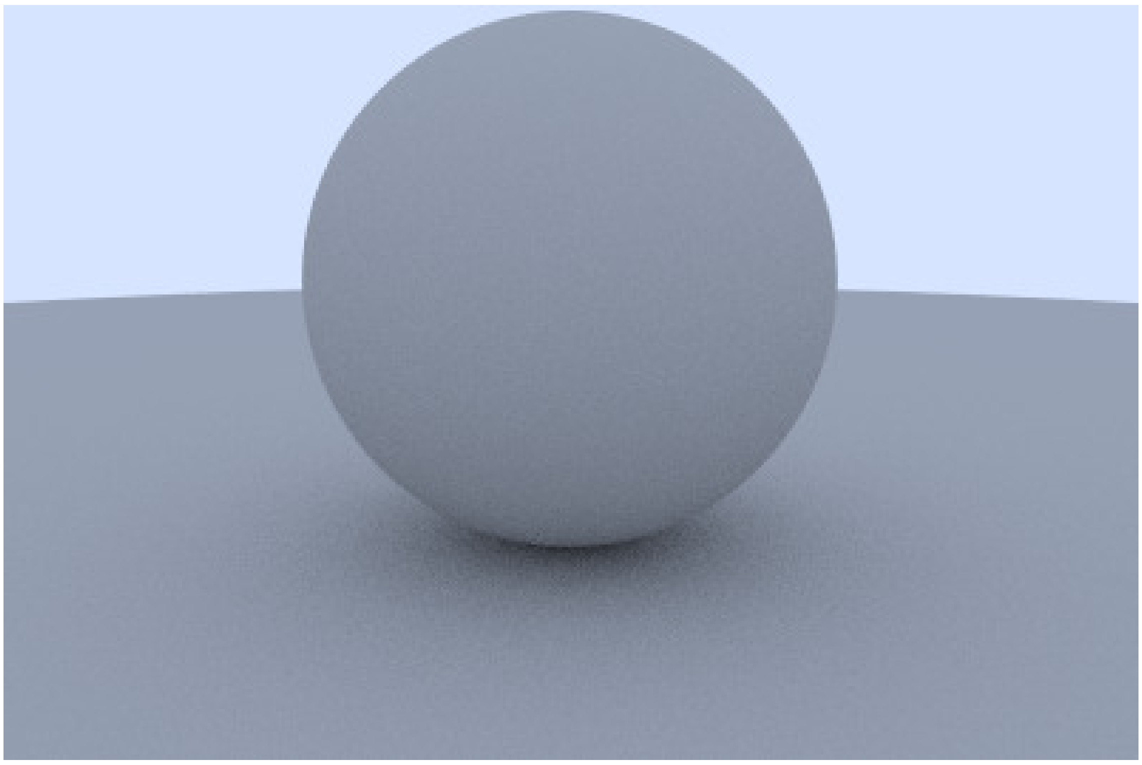


Figure 3: Here are two lambertian spheres (one acting as the ground) rendered using a diffuse scattering method.

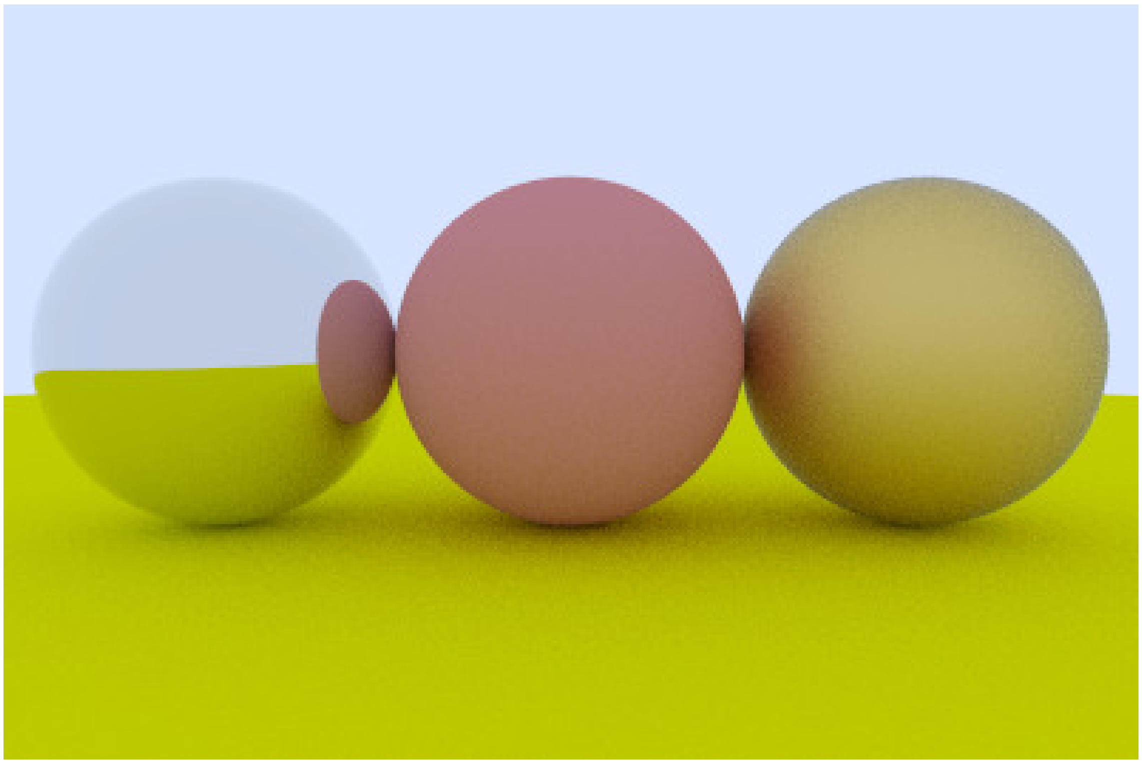


Figure 4: Here are two metal spheres and a diffuse sphere (center) on top of a large green diffuse sphere acting as the ground. The one on the right is with a maximum fuzziness value, and the one on the left has a minimum fuzziness value.

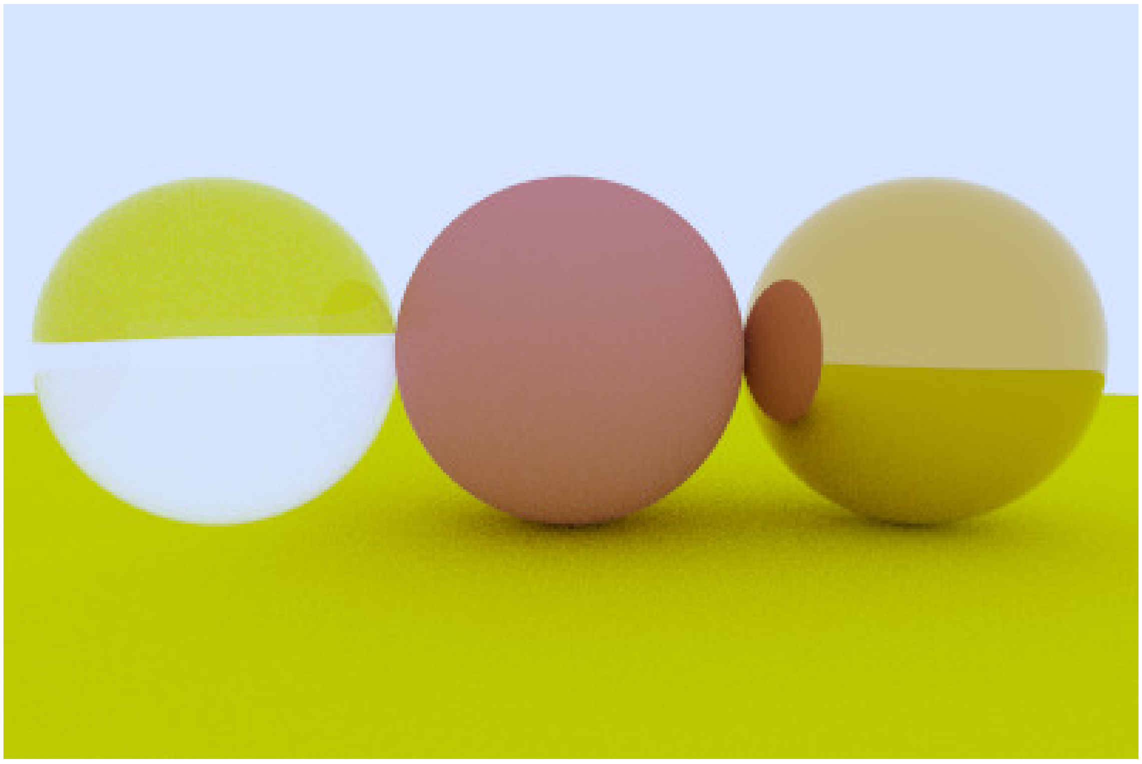


Figure 5: Here are three spheres on top of a large green diffuse sphere acting as the ground. The one on the right is a metal with a minimum fuzziness value, the one in the middle is a diffuse sphere and the one on the left is a glass sphere with index of refraction 1.5.

2.3.4 Lights

The ray tracer can also handle having internal lighting sources. The way it does this is by having an emitted property for all materials. It is generally a (0,0,0) vector for most materials, but for a light you can set it to emit values, which then ray_color interprets as increasing light rather than decreasing as it hits an object. This allows you to use the same objects defined before to light scenes. An example of this is shown in Figure 10.

2.3.5 Constant Density Mediums

A cool concept in ray tracers is the ability to simulate fog or smoke. This is done by letting the ray pass through a volume (given by an object) and allowing it to scatter at any point inside the volume. The scattering is once again similar to the Lambertian scattering, it picks a random uniform direction and follows the ray in the new direction. This code assumes once the ray has left the boundary of the volume it will not scatter anymore. An example of this is shown in Figure 14.

2.4 Textures

The next interesting thing to implement is to give objects a texture component. This differs from the previously discussed material properly as it does not have to do with how the light interacts with the object, but more to do with the color of the object. A simple example of a texture is a solid color. One can also easily make this a checkerboard pattern by using sines and cosines, as sine and cosines multiplied in 3 dimensions generates the desired checker pattern. This is shown in Figure 6. A more interesting texture is generating random noise. Ken Perlin invented a technique to generate random noise using psuedorandom numbers, and now much of random generation in programs is based off his original algorithm. I implemented this in my program by using hashed random vectors and interpolating between them in order to generate something that looks random. I can then use multiple summed versions to generate turbulent textures.This is shown in Figure 7. A last cool texture that I implemented was using random images as textures. These can be mapped

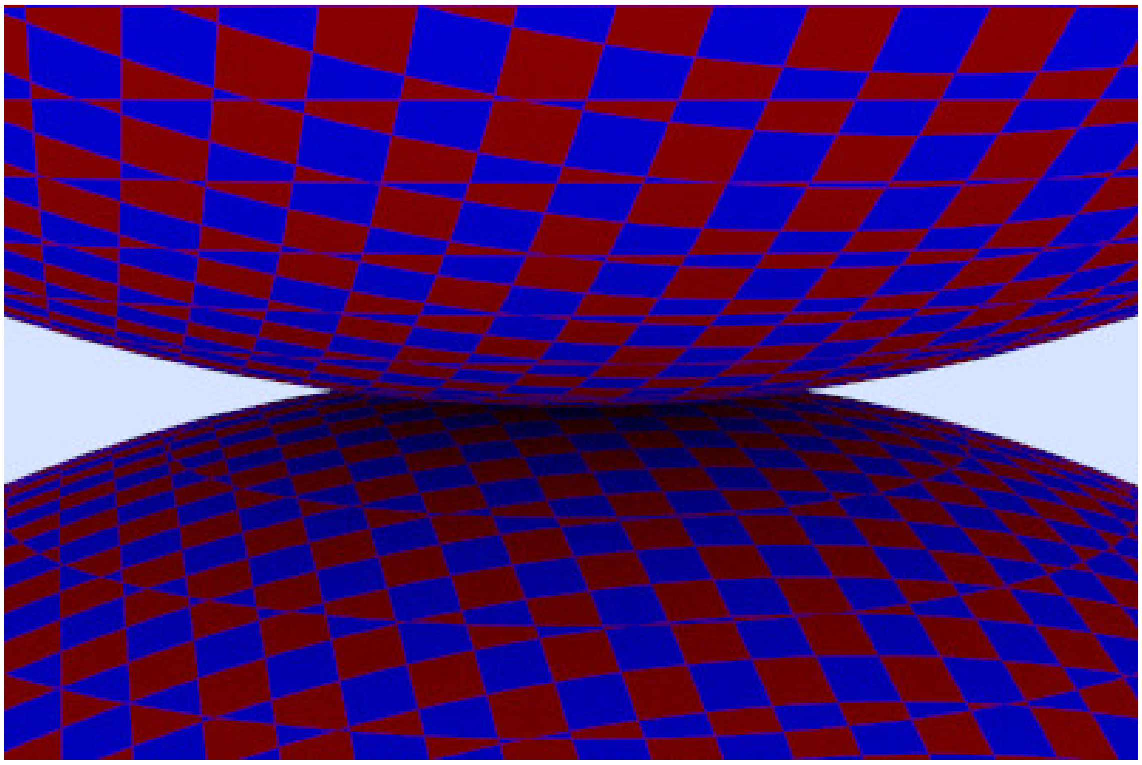


Figure 6: Two checkered spheres

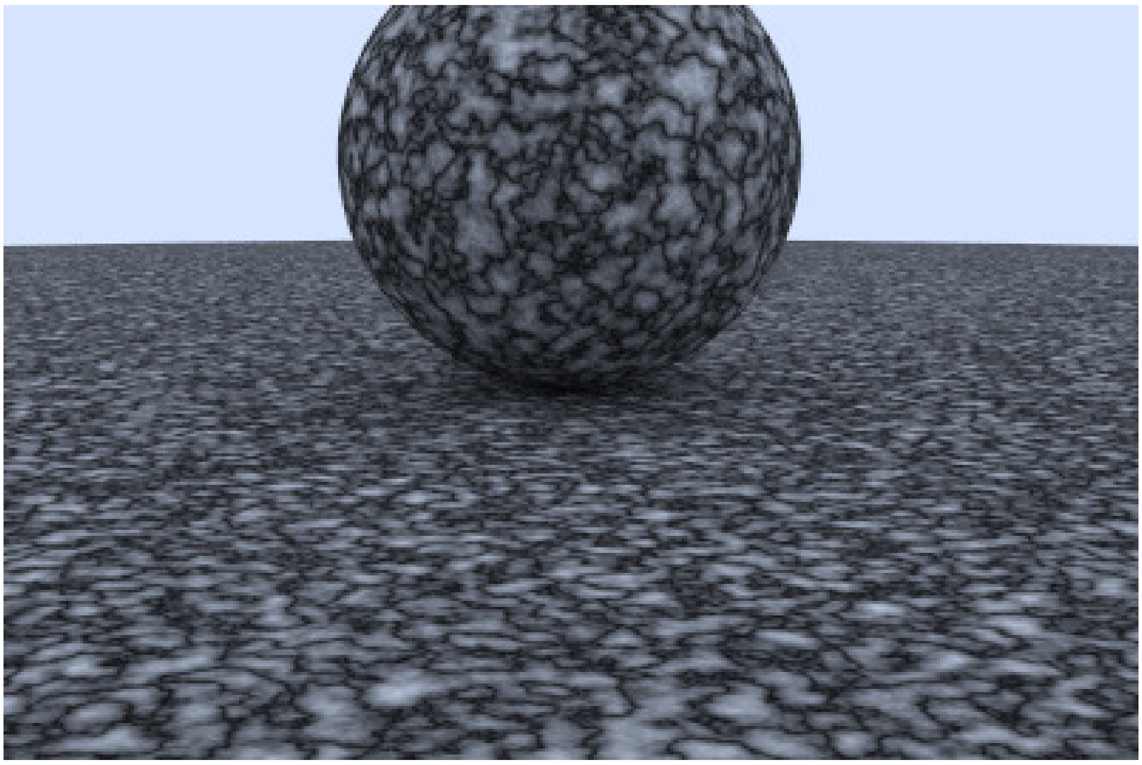


Figure 7: Turbulent Noise

to an object in a fairly straightforward way by breaking up the image into fractional positions, and then wrapping those around the object. So long as it is clamped around the bounds, it will map properly. Examples of this can be found in Figure 8, 10, 11.

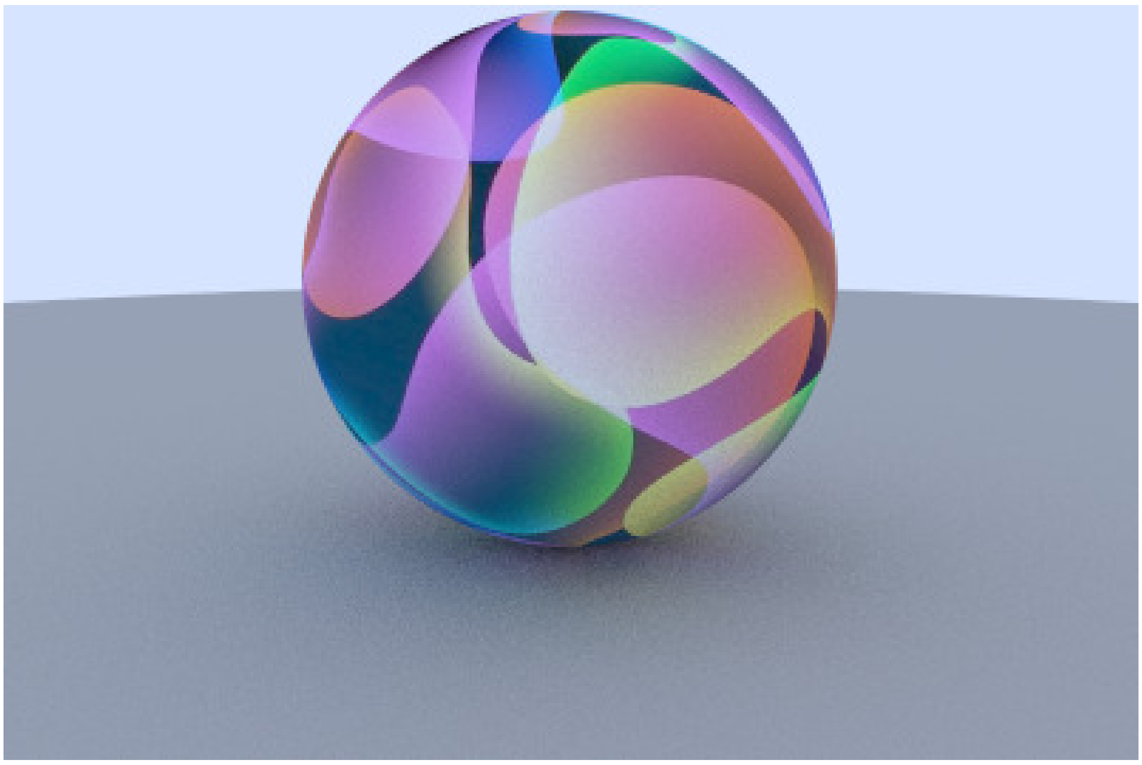


Figure 8: An random image I got off of a free wallpaper site mapped around a sphere

2.5 Bounding Volume Hierarchies

When there are lots of objects, this brute force ray tracing method tends to run slowly. The goal of this section is to illustrate a method in which that can be improved. This ray tracer works by generating a random ray aimed at a pixel, and then searches through the list of objects to check if any of them were hit. This search can be made smarter by considering the objects' bounding boxes. A bounding box is just a simple box that surround an object, and only exists virtually (i.e. it does not render). The idea is to create bounding boxes around all of the objects, and then cleverly divide bounding boxes hierarchically around groups of objects to create a search tree. That way, when this tree is searched, if it misses one node, it doesn't have to search any of its children, greatly speeding up renders with tons of objects. Thus each object has a bounding box method, and there exists a bvh node class specifically for creating a search tree.

2.6 Probability Density Functions

The final functionality I built into this ray tracer is noise reduction through probability distribution functions. The core idea is that in the previous implementations, small light sources create lots of noise, due to the fact that they aren't sampled enough. The previous implementation randomly sampled everything uniformly, what probability density functions allow for is to send more random samples in particular directions. If you just sample the light more, you will cause the image to be inaccurately bright, so you have to down-weight the samples to account for the oversampling. This is where probability density functions come in. Here I made a couple probability distribution functions, are a cosine probability distribution function, which uses the cosine of the angle of the ray to calculate the probability of where the light ray should scatter (very similar to how the random scatter was chosen for lambertian materials, except using a hemisphere instead of a sphere), and a object related pdf. The object related pdf allows you to send more rays at a particular object, to get greater accuracy, and then down-weights the sampling to account for extra brightness. These two probability density functions were combined in a so called mixture probability density function (which is just the weighted average of the two). Now with this I can pass in objects that I would like to sample more into the ray_color function, which allows for images with less noise.

3 Results

3.1 Images Rendered from the Initial Implementation

All these calculations allows for computers to create images from scratch, which is a pretty cool concept. I tested all of these ideas in several ’worlds’ the results of which are the following figures:

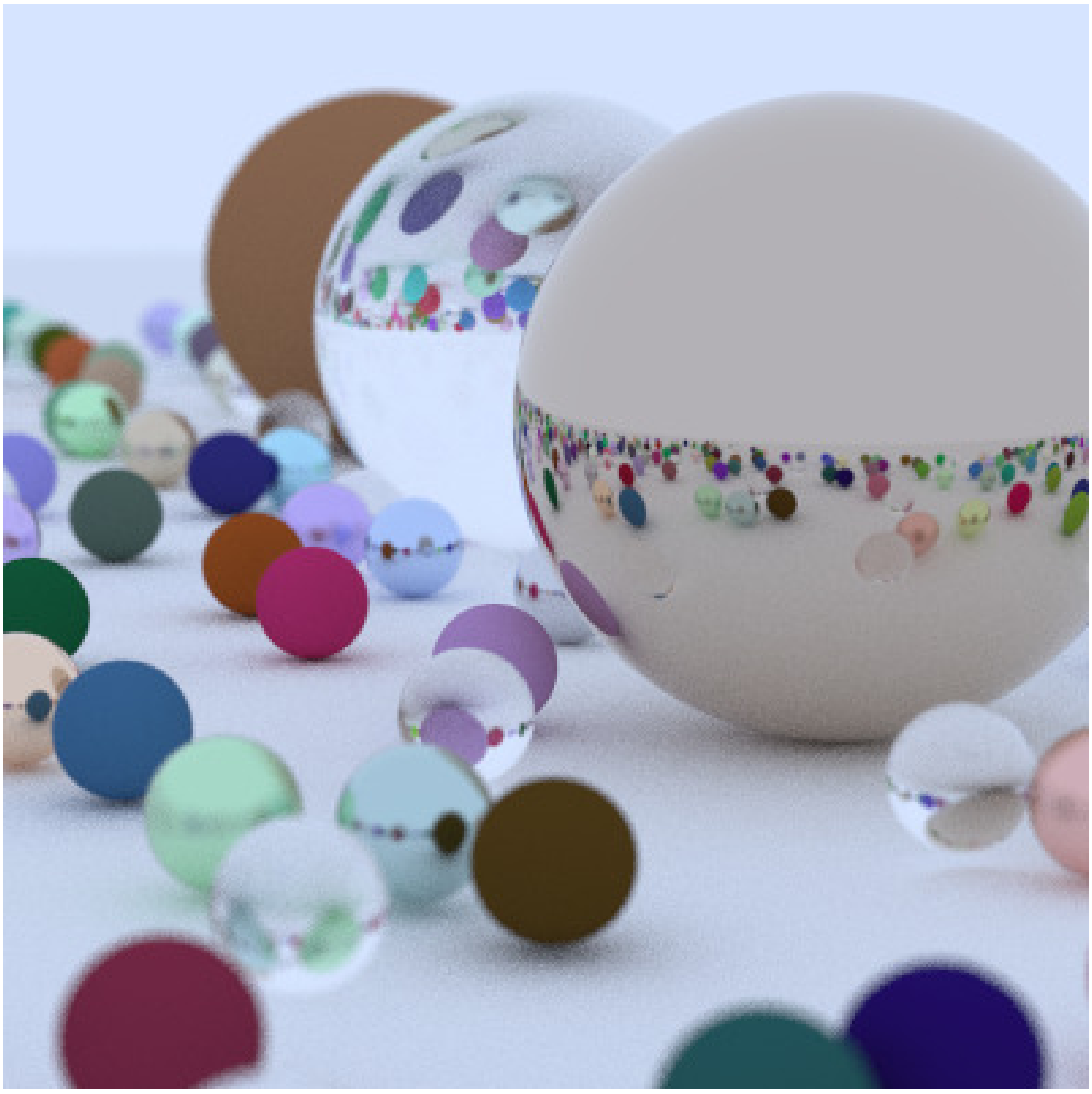


Figure 9: A bunch of random spheres

3.2 Comparing Images Rendered using the Probability Distribution Functions

3.3 BVH Performance

I also wanted to investigate the performance of the bounding volume hierarchies to make sure my search time truly become logarithmic.

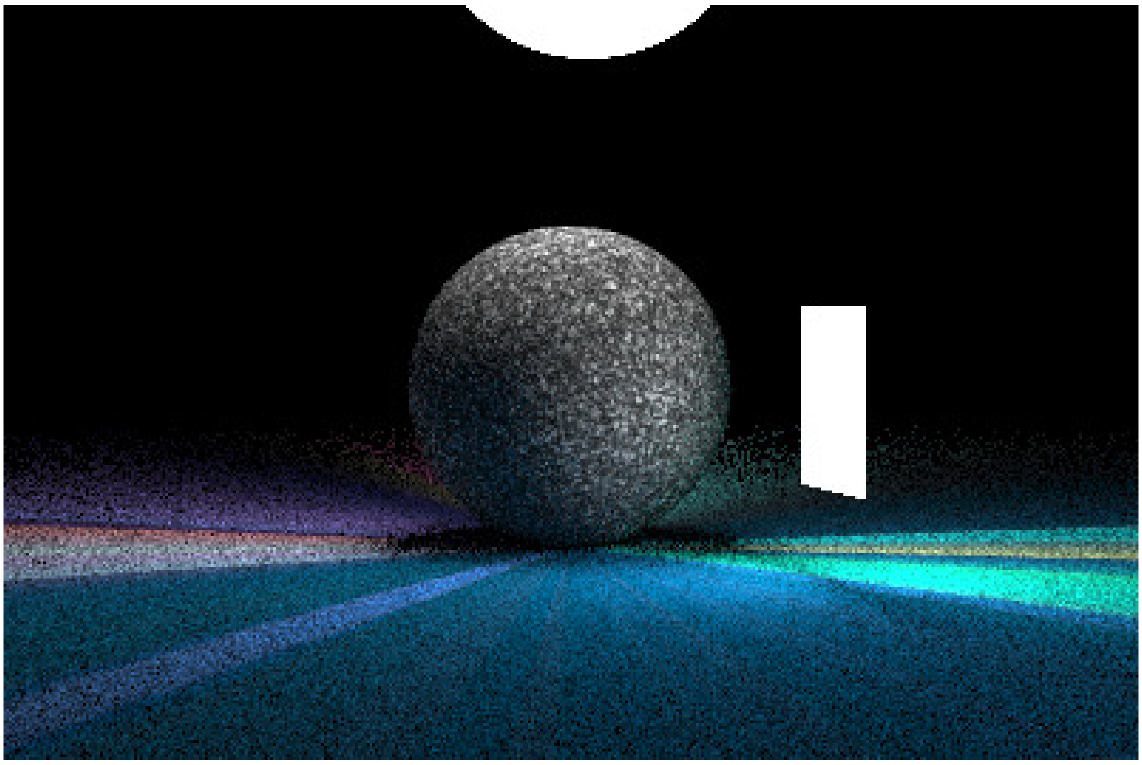


Figure 10: Lighting a sphere from inside the world

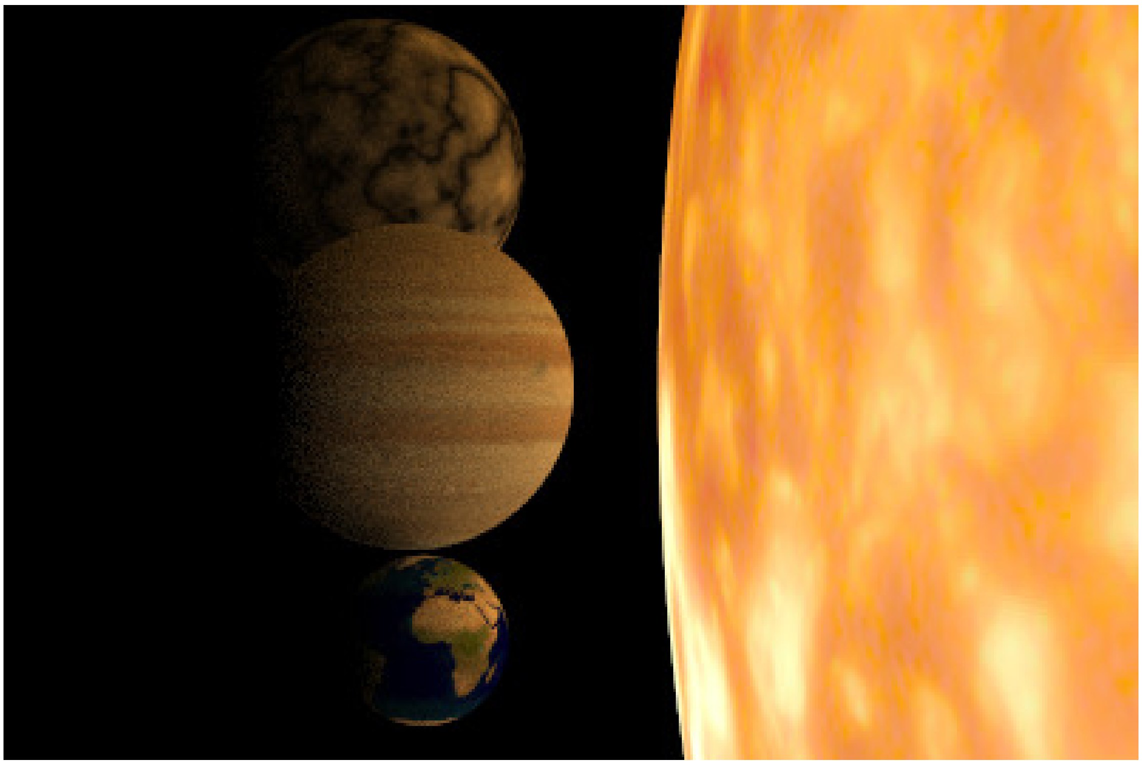
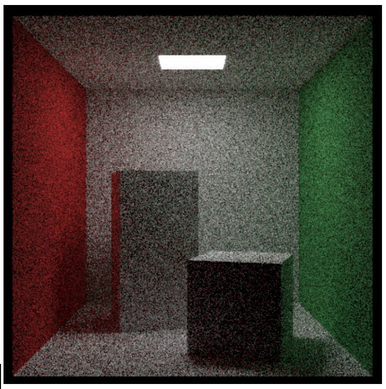


Figure 11: Using a an image as a light source to light other images.

4 Discussion

It is difficult to judge true accuracy of rendering methods without reproducing and photographing the exact render in the real world. However, I would say by comparing to previous work done

[]

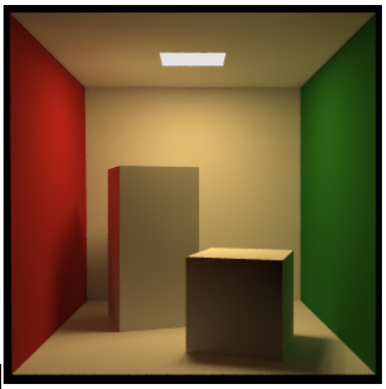
[]

Figure 12: The Cornell Box: a standard test used by scientists at Cornell when they were developing ray tracers was to compare a rendered image to a real life box where they could take an image and compare it to the rendered image. Here my render is on the left vs the real box on the right. The real box is from the wikipedia page the about Cornell Box

by Peter Shirley and the Cornell Box I can safely say that the initial implementation of the ray tracer that uniformly samples all pixels works accurately. It is also good to see that the new search algorithm scales logarithmically rather than exponentially (see figure 21).These results matched what I expected for the standard method. To properly test this ray tracer, I just made sure when I generated different worlds with different properties, that I got an image that made intuitive sense. I had the most trouble with implementing correct reflections upon rotating or translating objects, as I had to figure out how to keep track of the new normal vector of the shifted object in order to get a genuine reflection. When implementing the probability distribution functions, I noticed that there were some strangely bright sampling effects in the upper left corner of the Cornell box in probability density function version of the ray tracer. I believe that this is a sampling artifact, as it disappears when I sample uniformly.

4.1 Future

I would be interested in a couple paths for future exploration. The first would be: what happens if I relax the rigidity the ray, and treat it as if it was a wave? Could I simulate a quantum ray tracer interacting with quantum objects rather than macro objects? Could I simulate sound waves on a macro scale? I would also be interested in interactions with more complex objects, as I only had time to play with a couple. I would also like to implement something to speed this calculation up, which I think I could do by parallelizing the process of sending rays.

5 Conclusion

In this project, I built two versions of a ray tracer. The first is a general brute force ray tracer that sends out a ton of rays from the origin and calculates the interaction with objects of various shapes, sizes, and differing material properties. I also implemented different textures to allow for greater flexibility in the models I could build. This allowed for a good test of the classical behavior of light. I also built a probability distribution function implementation of the same ray tracer, that can send more samples at objects for better resolution in certain areas.

6 References

References

[1] Peter Shirley. Ray tracing in one weekend, December 2020.

[2] Peter Shirley. Ray tracing: The next week, December 2020.

[3] Peter Shirley. Ray tracing: The rest of your life, 2020.

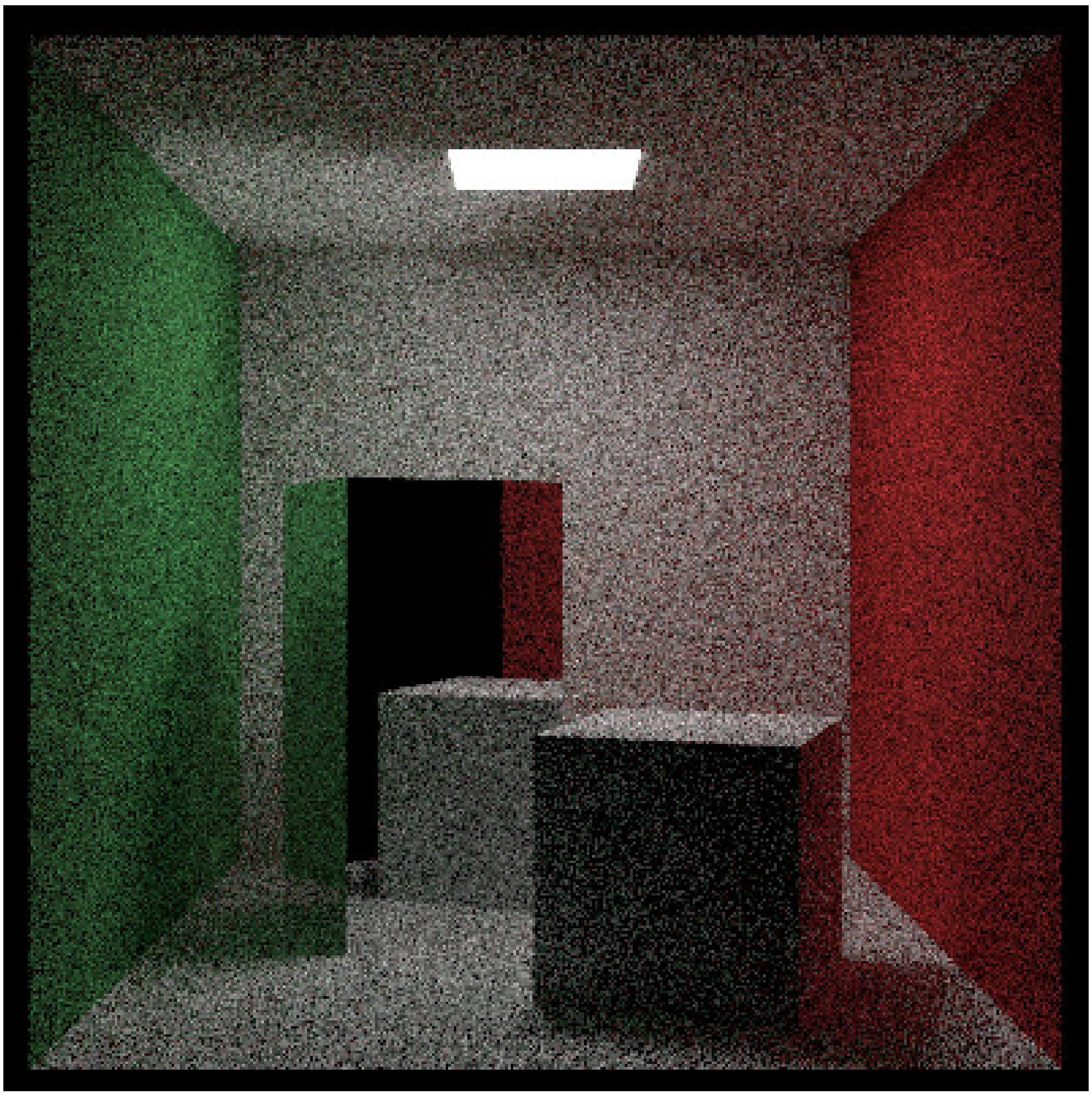


Figure 13: Using a mirror to make sure reflections still work if a box is translated and rotated. This was also used as a test case for comparing the probability distribution function implementation to see if noise was reduced.

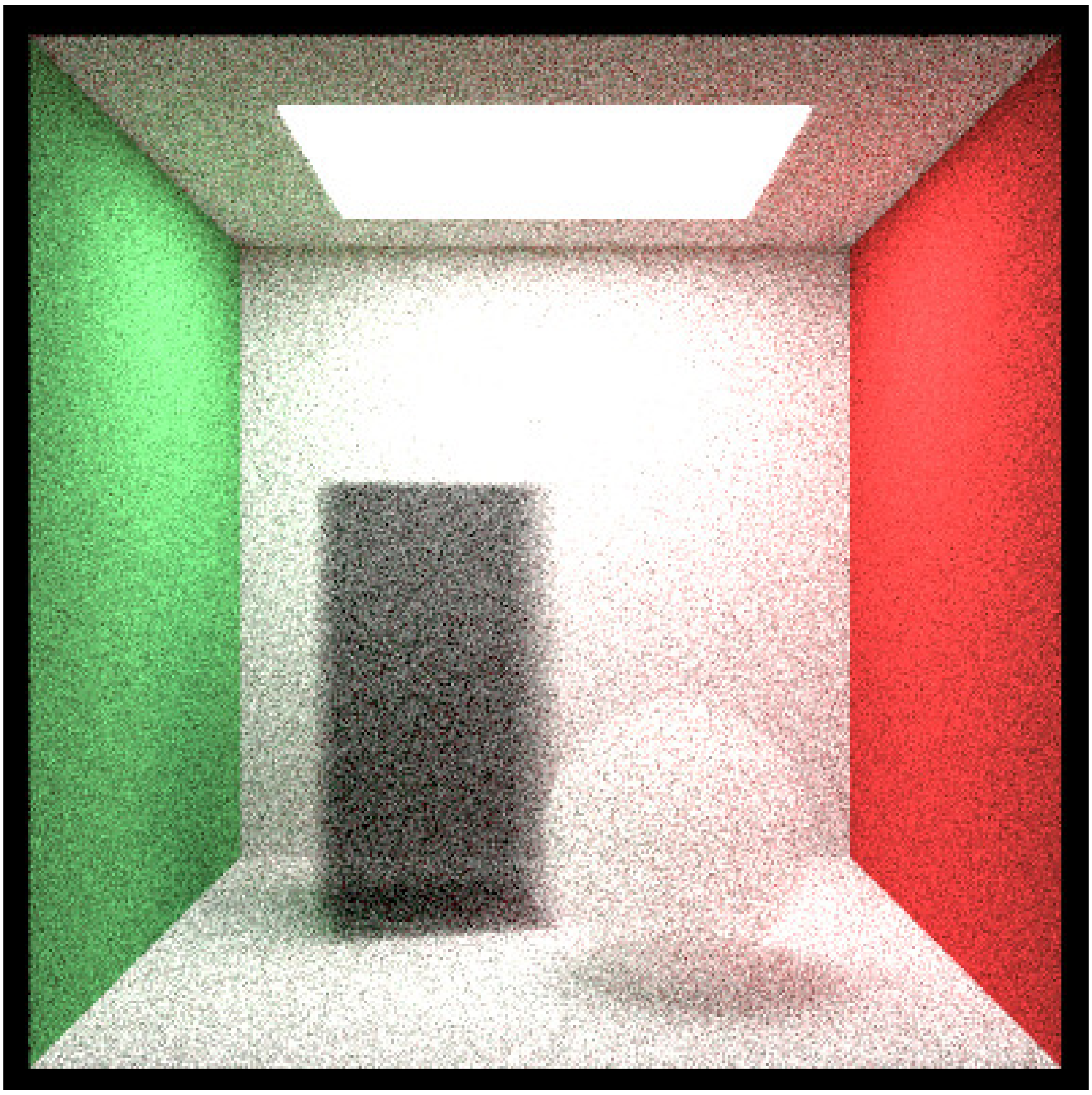


Figure 14: A box and ball made out of smoke

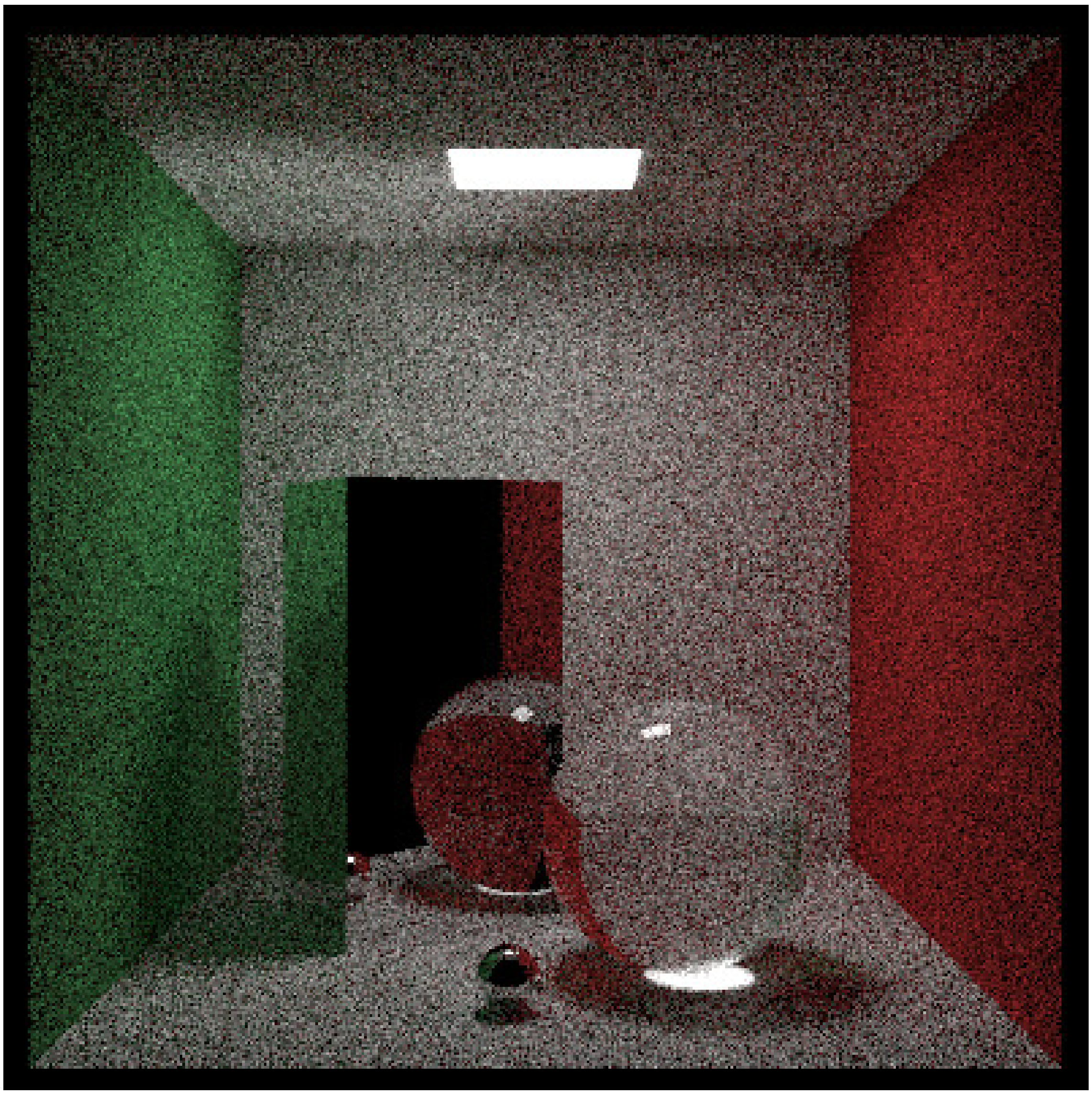


Figure 15: A mirrored box, a glass ball, and a smaller metal ball. This was also used as a test case for comparing the probability distribution function implementation to see if noise was reduced.

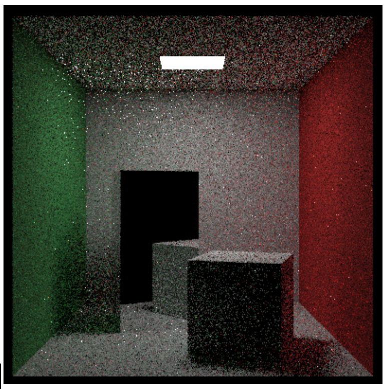
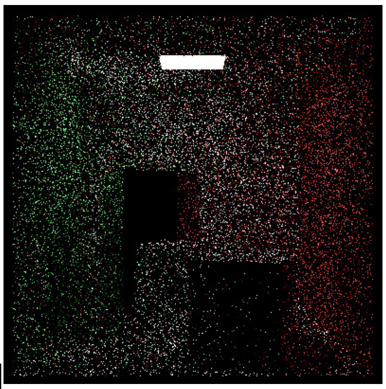
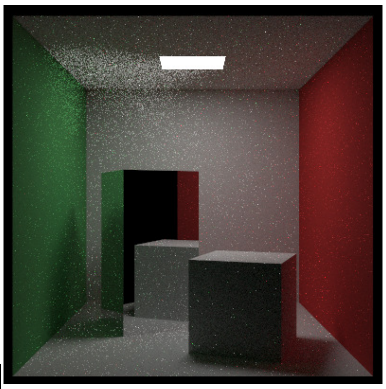
[] []

Figure 16: Depicted here is world 11, the one on the right is the standard implementation whereas the one on the left is the pdf implementation. Both images used 5 samples per pixel


[]

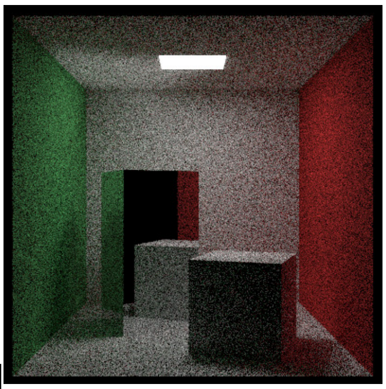
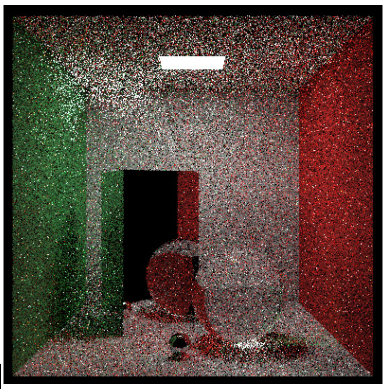

[]

Figure 17: Same as above, except both images used 200 samples per pixel


[]

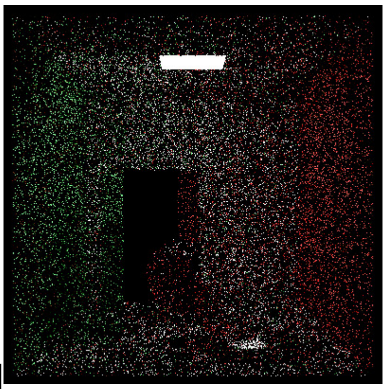
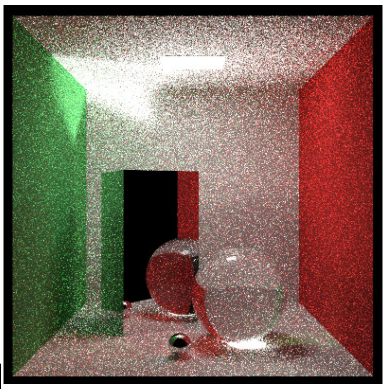

[]

Figure 18: Depicted here is world 13, the one on the right is the standard implementation whereas the one on the left is the pdf implementation. Both images used 5 samples per pixel


[]

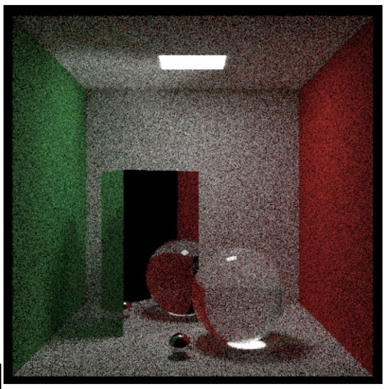

[]

Figure 19: Same as above, except both images used 200 samples per pixel

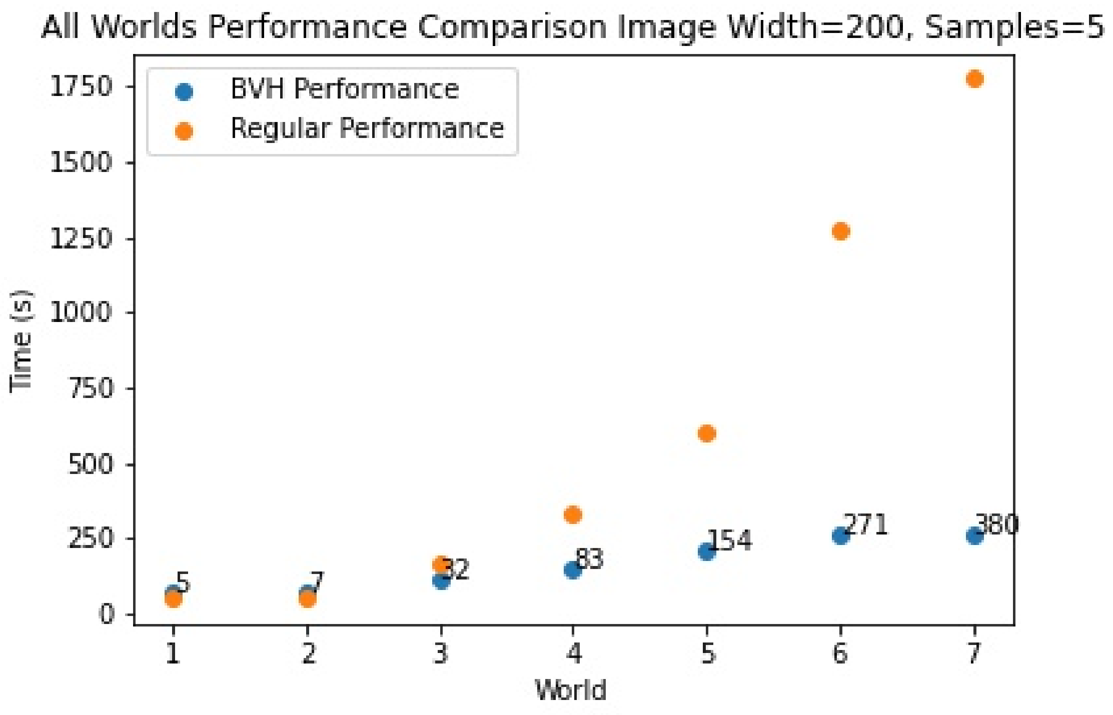


Figure 20: Here the number on top of the points represents the number of items in each world. These were all calculated at 5 samples per pixel.

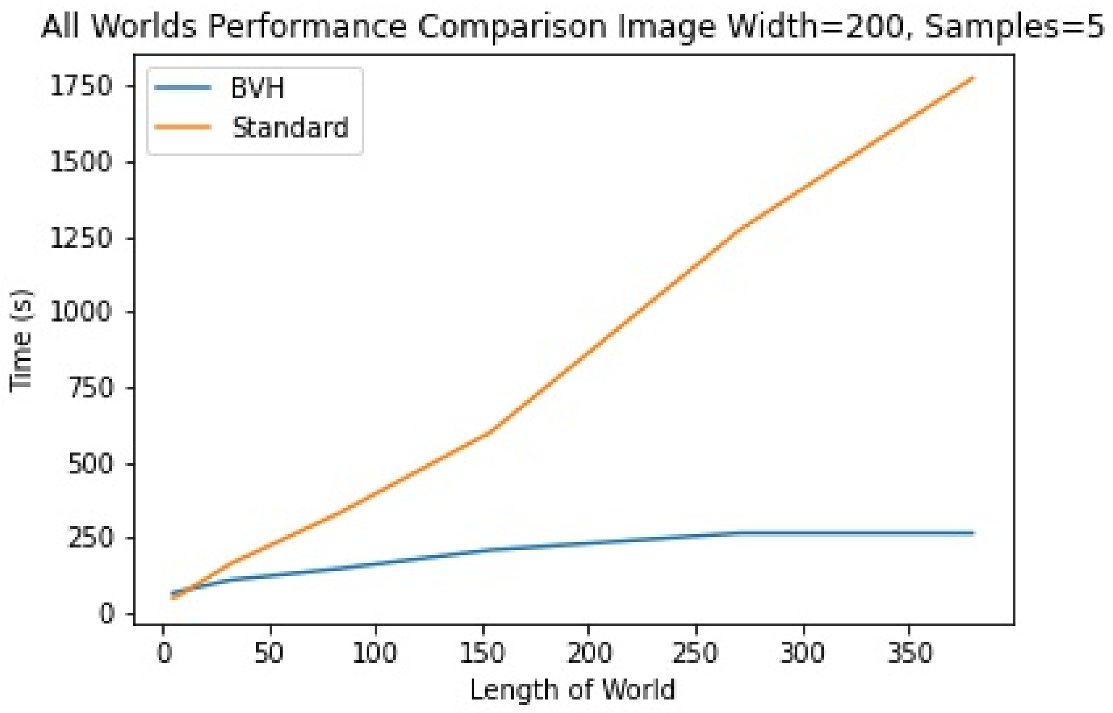


Figure 21: