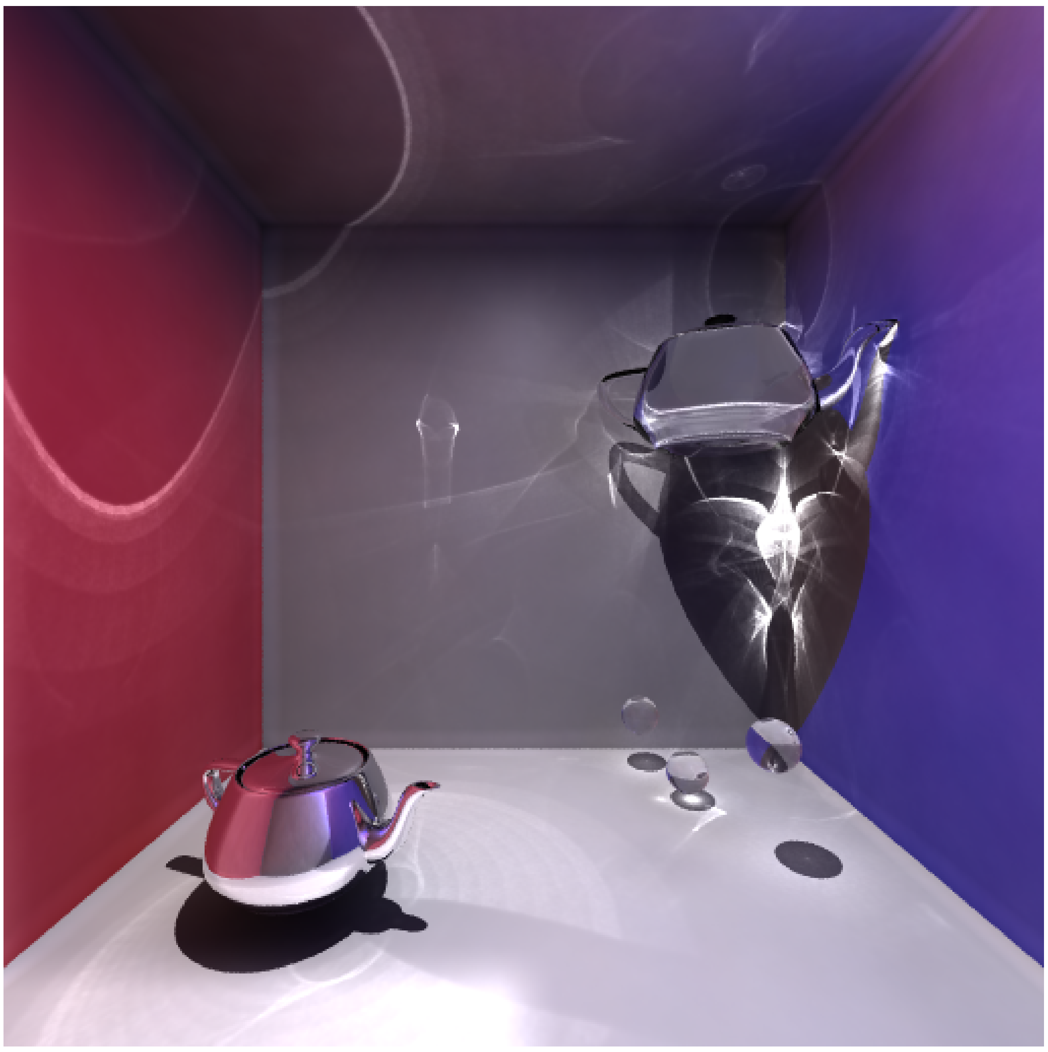
Advanced Computer Graphics

Raytracer Implementation

December 8, 2023



1 Global Lighting

1.1 Introduction

Reflection and refraction are global lighting effects that the raytracer was missing. Global lighting materials are those that can interact with other materials and objects in a scene. They include reflective materials, that can bounce light from object to another like a mirror, and refractive materials, which are partially-transparent materials where light changes speed upon entering them, leading to a distortion effect. It is argued that raytracing provides one of the most eloquent ways to render these effects (Cook, Porter and Carpenter, 1984), as we simply raytrace more rays, so our raytracer should support rendering materials such as metal mirrors, glass and water.

1.2 Implementing Reflection

For this implementation, we have separated global lighting into two materials that inherit from `GlobalMaterial`. This was really just to make each file smaller, but allows us to have a separation of metals (purely reflective) and refractive materials (also called dielectrics, (Peter Shirley, 2023)). We will see later on that reflection and refraction aren't truly separate, but related based on the viewing angle.

The framework code provides a built-in method on the `Vector` class, which takes in an incident ray, and provides the reflected ray. The vector that calls the `reflection()` method is the normal vector of the surface. The dot product is used to find $cos(\theta)$ (the vertical component of a normalised vector), which can be doubled and negated to get how far above the surface we should be, when added to the incident ray (from the origin or hit point).

In the `compute_once()` method of our reflective material, we use the `reflection()` method to find the next direction we should send another ray in. We send another ray as it means we can just recurse using the same `raytrace()` method, and that will handle inter-object decisions automatically. We then need a recurse limit, set in `render_settings.h` that will stop infinite bouncing off of (e.g.) two opposite mirrors.

If we implement the algorithm like this, it works out mathematically, however we get a dotty image, as shown in Appendix **A.1**. This issue occurs as floats have a limited precision. This means that our ray could end up beneath a surface on a hit, and our next ray would just hit that same surface immediately. We can fix it by shifting the new ray a tiny amount along it's direction. There are some renders of compound reflection in Appendix **A.2**.

1.3 Implementing Refraction

For refraction, we write code that computes the refracted ray direction, based off of the equation derived by Solomon (2020b). The equation is derived from Snell's law, which describes how the index of refraction (the ratio of the speed of light in two media) affects the angle of the ray towards the normal. We know what the index of refraction is for common materials (e.g. air-glass = 1.54 / 1) and we pass that into our constructor.

The index of refraction will change depending on whether we are entering or leaving an object. The material takes in an index of refraction, however we need to use the inverse of it if we are *leaving* the object. The easiest way to keep track for `PolyMesh` (and other) intersections, is to attach the `entering` property to the ray itself, and just flipped it when we refract. There is a downside to this approach that we discuss later.

Before we can compute the transmitted ray direction, we need to make sure that a solution does exist by checking that the radicand is greater than zero. If it isn't, we reflect on the inside of the surface, undergoing *total internal reflection*. This works just as reflection does. Refraction also needs a recursion limit (as you could imagine four refractive spheres creating a loop), and also suffers from spots, so we use our shifting trick again here.

1.4 The Fresnel Equations

With a transparent material, such as glass, we can get reflections on the surfaces if we look at them through steep angles. The Fresnel equations describe how much light should be reflected and refracted, based on the angle of incidence of the viewer. Initially when implementing them, there was a bug causing spotty artefacts. This was because kr, and kt were properties of the material that would get slightly updated every recursive call. This was fixed by making them local. In this implementation, we still use the coefficients passed in to allow adjusting the final weights to get more customisation. There are a collection of refraction renders in Appendix **A.3** shows off refraction with the Fresnel reflections.

1.5 Implementation Limitations

The first limitation is that this raytracer can't render "fuzzy metals", as described by Peter Shirley (2023), but that would be easy to add; we can slightly offset the direction, blurring the reflection by some "fuzz" amount.

The second is that for nested media, refraction wouldn't be correct, as we use the reciprocal to determine the index of refraction, assuming that the rays we send will always enter and leave via air, where we approximate it as 1. To fix this, we could add a stack to our ray class, keeping track of all the changes.

Finally, when light refracts, the different frequencies all change speed by different amounts. This is how we can get rainbows, where light refracts through water droplets (Royal Meteorological Society, 2020). This is only possible during photon mapping, as backwards raytracing can't determine the origin of refracted light on diffuse surfaces.

2 Quadratic Surfaces

2.1 Implicit Surfaces

Implicit surfaces are surfaces that are modelled using mathematical equations, where a point lies on the surface, if that point solves the equation. Quadratic surfaces are a case of implicit surface, defined by a 4D matrix that is symmetric across it's diagonal, where a point (x, y, z) is on the surface if it solves the expanded-out, general form given by (1). It is quadratic as the highest exponent of x_p, y_p, and z_p is 2. A sphere is an example of a quadratic surface, as the equation of a sphere is a form of $x^2 + y^2 + z^2 = 0$.

$$ax^2 + 2bxy + 2cxz + 2dx + ey^2 + 2fyz + 2gy + hz^2 + 2iz + j = 0 \tag{1}$$

Quadratic-ray intersections can be solved just like sphere ray-intersection equations can be solved. We can use algebra as we know all the points that lie within the ray, and all the points that lie within the surface of a quadratic. Setting these two equations equal gives us all the points that lie on both the quadratic and the ray, and provides us with the two intersection points. This implementation of a raytracer discards the tangent points, as it features anti-aliasing it which smoothes edges anyway.

For testing, there is an interactive viewer of quadratic surfaces (Davis, n.d.). This website uses different axis to us, so we can apply the transformation that swaps the y-axis and the z-axis, which is just those two columns swapped in the identity matrix.

2.2 Transforming Quadratic Surfaces

We apply transformations by simply updating the definition matrix. This implementation saves the definition, so it is simply some matrix multiplication. There's a difference between the transform supplied in the short note, and the transform by House (n.d.) that we've used, which is that House (n.d.) inverts the transformation matrix.

Applying the inverse makes our quadratics behave how we expect transforms to work. Without the adjustment, they do the opposite of what we would expect. Scaling 3 times would shrink by 3 times, and so on.

2.3 Modelling Shapes

The quadratic surface is useful, as we can model a large variety of shapes using it. These shapes are far more expressive than the simple sphere and cubes, and can be used to create very detailed objects with constructive solid geometries (CSGs). To change the shape of the quadratic, we can start of with the simplest cast of a cylinder, where only two of the terms are non-zero (e.g. $x^2 + y^2 = 0$).

We can then slowly introduce and scale terms from equation (1) to get more and more complicated shapes, such as cones, ellipsoids, parabola, and even planes. For example, scale a squared term for ellipsoids, negate some terms for hyperboloids and so on. See Appendix **B.1** for examples. Appendix **B.2** demonstrates some examples of using quadratics with CSG and global lighting. All quadratics have undergone some transformations for visibility, particularly the transform that swaps the y and z axis, to compare against the geogebra page (Davis, n.d.).

3 Accelerated Ray Tracing

3.1 Acceleration

For this raytracer, a further feature that was implemented was the acceleration of the raytracer. This feature was chosen as it is almost essential to speed up the photon mapping step, which would be very inefficient without it. Naïve raytracer implementations will do intersection tests for every primitive, for every pixel, for every sample. This is a terrible approach and scales poorly, and we even waste a ton of computation on objects that may not even be in our render. The problem is that a complex mesh, such as the teapot, may not even be visible, and each ray will do 6000 intersection tests, testing each of it's triangles. For a 512×512 image, this is over 1.5 billion wasted intersection tests on triangles.

3.2 Bounding Volumes

Instead of testing each primitive individually, we can just test the teapot as a whole, and only then decide to test the triangles, if the ray intersects with the teapot. We can't test the teapot on it's own, however we can wrap the teapot in an enclosing, invisible object, that is faster to test against. This raytracer implements axis-aligned cuboids, as they are easy to create, given the minimum and maximum points of an object.

`compute_bounds()` in this implementation, is only ever called upon the first intersection test, as transforming a object unfortunately doesn't also transform it's bounding box (due to the axis-alligned assumption), so we only want to compute the bounds once we know the object will no longer be transformed. The trade off to this simple approach, is that we can't have an automated bounding volume hierarchy, where bounding boxes contain other bounding boxes, as they're all created at render-time, as opposed to scene building-time.

The intersection tests this implementation uses, are inspired by the pseudo-code proposed by Solomon (2020a), however I had to come up with the implementation details and handle the edge cases myself. Solomon (2020a) also proposes an efficiency trick, to precompute the reciprocal of the ray direction beforehand, but this wasn't implemented in order to keep the default constructor available and safe without adding accessors/mutators everywhere. The intersection test works by finding the smallest, common range of t values (ray parameter) that our ray lies within for each axis. If one exists, we have our hit points.

Finally, there is no reason why we can't bound other objects, however all the other objects are implicit surfaces, and are only intersection tests themselves. To avoid thinking about infinities, and detemining whether its faster to bound (e.g.) a sphere, or leave it unbounded, we only bound `PolyMesh` objects.

3.3 Bounding Volume Hierarchies

Bounding objects are useful, however for complex scenes, it can also be worth implementing a hierarchy of bounding objects. This means allowing bounding objects to hold other bounding objects. We can then create a rough binary tree for database (scene) lookups, improving intersection searching from $\theta(n)$ to $\theta(\log_2 n)$, for n objects in the scene. This is great, however for small values of n (roughly < 10), the speedup is marginal. This means that as our scene isn't very object-heavy, the bounding volume hierarchy isn't worth the development time. Instead, we can get far better results with another acceleration technique.

3.4 Parallelisation

This implementation includes generalised, n-way parallelism for the raytracing step. To balance work evenly, each thread takes every (mod $n)^{th}$ row, where n is the number of threads. This is far better than the initial approach which separated work unevenly (just chunks). The effect is shown in Table 1. This was a lot cheaper (in developer time) to implement, as 3D rendering is one of the well known "embarrassingly parallel" applications (Arm Developer, n.d.).

1	2	3	4	5
1	2	3	4	5
1	2	3	4	5
6	7	8	9	10
6	7	8	9	10

Table 1: Work division per colour (thread)

There were a few bugs with the initial parallelisation. There had to be locks on the hit-pool, kd-trees (for photon mapping), and we had to make sure shared objects, such as materials, weren't updating their values, but using local copies in method calls.

3.5 Speedup Evaluation

The overall speedup of both bounding boxes, and parallel rendering is incredible. Parallel photon-mapping however, can actually slow down the implementation as the main bottleneck is kd-tree insertion, millions of times, which has to be locked. A smarter implementation would give each thread a local buffer of `PhotonHit` objects, and then at random, lock the kd-tree and empty the buffer, so threads are unlikely to wait at the same time.

In summary, we were able to achieve infinite speedup using just the bounding box on the teapot (when it's far enough away), and almost a 300× speedup for a tiny teapot when we include parallelism. In terms of real use, we can photon map and render the Cornell Box used by Jensen and Christensen (2000) with reflections, refraction, and global illumination, in just over a minute (80 seconds), building the photon map in 2.8 seconds, with 100,000 photons and realistic importance sampling and nearby point counts.

The full table of speedup values and their associated visuals, can be found in Appendicies **C.2** and **C.1** respectively, figures being run on an 1.4GHz Quad-Core Intel Core i5 with hyperthreading (macOS).

4 Global Illumination

4.1 Global Illumination Effects

The goal of our raytracer in this step is to introduce global illumination. The main idea behind global illumination is that light can bounce between surfaces, so even areas that have no direct lighting, can receive indirect illumination from other surfaces. This gives an effect known as *colour bleeding*, where the colour of one surface can leak onto other surfaces, shading them.

Another goal of global illumination is to render *caustics*, where light focuses sharply due to reflection and refraction mapping multiple incident light rays to very similar points. Backwards raytracing cannot handle these effects, as it cannot follow specular to diffuse light paths. Renders showing these effects can be found in Appendix **D.1**.

There are various techniques for implementing global illumination, including the radiosity technique proposed by Goral et al. (2002), Monte Carlo methods (such as Metropolis Light Transport proposed by Veach and Guibas (1997)), and Bi-Directional Path Tracing, proposed by Lafortune and Willems (1998). Jensen (1996) claims that Photon Mapping simplifies the representation of illumination, and converges more efficiently with accurate caustics. One of the downsides of photon mapping is that we can get blurrier caustics, as we are interpolating values, however there are methods to combat that.

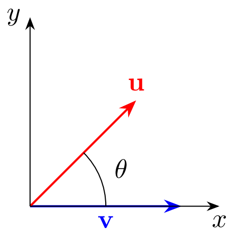
4.2 Photon Map Creation

The forwards pass is a pass of our raytracer where rays are emitted directly from the light source. We do this to gather information about how light works in our scene, and then use that information in the second pass (backwards raytracing). We call the light rays 'photons' in this implementation, and my `Photon` class inherits from the ray class. Jensen and Christensen (2000) state that you can use any type light source to emit photons from, however for the Cornell box we implement a point light source with a position and direction. This allows us to test if photon mapping works, by checking if the ceiling is lit, such as in Appendix **D.1**.

Given a point light source, we now have to decide how we will emit photons. In reality, the flux of photons changes based on the direction the photon was emitted, compared to the direction of the point light source, following a cosine distribution. However as we are sampling, Jensen and Christensen (2000) state that we can send out a number of photons proportionally to their flux, and keep the intensity of each as the intensity of the light. This reduces work, as we get more 'high impact' photons, and less 'low impact' ones.

To do this, we have a `get_random_photon_direction()` as a virtual method that lights must implement. In his book, Jensen (2001) describes a way to generate cosine weighted vectors in a hemisphere (we use them later too). We can borrow this idea, but we must covert from polar coordinates to a vector, and then once we have that vector, transform it from a local coordinate space, to world space around the hit normal.

To do this, we can use Rodrigues' rotation formula, which rotates a vector around any axis that we choose (Mebius, 2007). You can imagine if you wanted to rotate $\vec{v}$ to $\vec{u}$ in the diagram, that we would rotate $\vec{v}$ θ degrees anticlockwise, around the axis coming out of this page. If we use the cross product between 'up', and the hit normal, we find a perpendicular vector describing the axis that we should rotate about. Theta can be found by taking the inverse cosine of the dot product between the vectors. Now that we have a direction to our photon, we emit it starting from the light source, and use the same `scene.trace()` method we've been for raytracing.



Unfortunately, the code for Rodrigues' formula wasn't working, so we have to come up with an approximation. The simplest (and fastest to implement) way to do this is to generate a uniformly random vector in a sphere using rejection sampling, normalise it, and then add a normalised bias term in (see `random_in_hemisphere(bias)`). This is similar to if you were to generate random vectors from the origin O of a unit circle that had been translated up by 1. It somewhat approximates the cosine distribution, at the cost of worse *ambient occlusion* at edges.

If we don't hit anything, we discard the photon. Jensen and Christensen (2000) propose that we build a *projection map* to handle the case where are scenes have sparse geometry and many photons will go to waste. For the Cornell box, this is not the case, so this raytracer doesn't implement that step. When we do hit a surface, we check to see if the photon came directly from the light source, via the `photon.from_light_source` boolean.

We use this property to determine whether we want to include the `PhotonHit` into our photon map. The photon map for now is some data structure that contains all the hits each photon has made through it's journey. The reason we don't include hits directly from the light source, is that we already do a direct illumination pass (the raytracing), by including it we would be doubling that effect, and then adding complexity, as we wouldn't save the colour of the photon, but rather the diffuse component of the surface it hit.

4.2.1 Photon Handling

When we hit a surface for the first time, we need to decide what happens to that photon. In reality, many photons will be shot out, based on the material properties. Some photons will reflect in a specular way, not picking up any colour. Some will be transmitted through a surface. Some will be absorbed by the surface, and some will undergo Lambertian reflection, where they pick up the diffuse component and get reemitted in a cosine-weighted hemisphere. This is another opportunity where instead of sending millions of weak photons for each hit, we could do a single photon with the full intensity, probabilistically, and use the same approximation as we did for emission.

We implement the Russian Roulette algorithm (Arvo and Kirk, 1990), which just randomly decides what happens to our photon based on the material properties. All `Material` classes have an `interpret_photon(...)` method which decides what happens to a photon based on the material it hit. The Phong material with a high diffuse component and a low specular component will be more likely to diffuse reflect than specularly reflect. This is true in general, however when we compute the probabilities as Jensen and Christensen (2000) propose, we sum the colour channels. This means if our blue material has diffuse component (0, 0, 0.6) and specular component (0.2, 0.2, 0.2), they will have the same weighting.

In C++ we can generate random numbers uniformly using the built-in `uniform_real_distribution` type. As we are dealing with random numbers so often, we can extract the random number generation into it's own class, called `RandomFloatUtilities`. The methods are static in this class and can be used from anywhere in the program. We can also move our method for generating random vectors there.

An alternative approach to everything mentioned above, is to directly use a Bi-Directional Reflectance Distribution Function (BRDF). This describes how much light leaves a given direction from a point, as a function of all the light that contributes to that point. Initially, a goal for the raytracer was to include BRDF compatibility, however it never ended up getting implemented. This means that for Photon Mapping, we just adapt our current materials to handle photons using Russian Roulette. The `CompoundMaterial` is not implemented, as we would have to iterate through all materials, get their diffuse, specular, and transmissive weights (which would mean all subclasses need all of these, or dynamic type checking), and then perform Russian Roulette.

4.2.2 Photon Storage

Now that we've reflected or refracted off a surface, the next (and subsequent) hits do need to be stored. Jensen and Christensen (2000) suggest using a KD-tree as an efficient way to do lookups, which will be essential for the second pass. The KD-tree we are using (Burk, 2017) implements n-space partitions, based on the initial `min` and `max` values passed in. Initially there was a bug where these were set to `-INF` and `INF`. This would cause exponentially slow renders, until it was realised that these values should wrap the scene as tightly as possible, making small renders almost instant. This is because the KD-tree doesn't perform balancing on each point insertion, but just splits along axis equally.

This implementation only stores the intensity and position. Jensen and Christensen (2000) also save the direction to determine where a photon came from. This is because a photon could bounce around and hit the back of a wall (a plane) and we would want to check and exclude that photon. In our scenes this isn't possible as we use only 3D objects inside the box, and the box is fully enclosed, and we can use far less memory avoiding it.

At this point it is also useful to make the distinction between caustic photons, and diffuse inter-reflected photons. Caustics should be sharp, and colour bleeding should be smooth, however if we had only one map, these two goals would be contradictory. Jensen and Christensen (2000) say that we should have two separate maps. The high-resolution caustic map, and the lower-resolution diffuse map. The diffuse map doesn't need to be sparser, but it can be as an optimisation. The caustic map must be high enough resolution for sharp caustics.

Another technique we could implement, would be to optimise shadow ray checking by storing shadow photons. This could be done by having another map, the shadow photon map, and performing lookups on it rather than sending out shadow rays. When a photon hits an opaque surface, we could send it through and record the next surface hit as one in shadow. This optimisation is a useful one if the raytracer is sampling many shadow rays. This raytracer doesn't implement it, as we only send out shadow rays once, as there is no area light.

4.2.3 Importance Sampling

To decide which map a photon should be saved in, we have a `bool caustic` field on the photons. This is set to true on the next photon, after a photon is reflected or refracted. We also want to send more photons that will be caustics to get the higher resolution caustic map, without increasing the resolution of the diffuse map. Importance sampling is the idea of taking more samples when something important needs to be measured. For us, if a photon becomes a caustic, then it's likely that other photons following that path will also be caustic, so we return early, and then send a bunch more photons in that direction.

This is implemented through a loop in `Scene::parallel_build_map()`, where the number of bonus samples we take is determined in `render_settings.h`. When we initially send the photon, we pass the reference in, which allows us to pass back the final photon at the end of the loop. If the final photon was a caustic, send a bunch more, with slightly offset direction, but keep a track of how many of those end up being caustics in case we got a false positive and can stop early. We also have an `is_bonus_sample` field, as we don't want to add any more diffuse photons to our map. The downside to this implementation, is that if a photon becomes a caustic, it's path must end there, meaning we can't model caustics that cause other caustics through a high amount of specular reflection on a Phong material.

Now that our photon map is created, we can visualise it as shown in Appendix **D.2**. The photon map is actually viewer-independant, and we can actually cache it, and reuse it (if our scene doesn't change) for multiple camera angles. This implementation unfortunately doesn't do that.

4.3 The Raytracing Pass

The raytracing pass uses all the information recorded, to get a more accurate sense of what colour we expect to see in each pixel. It does uses interpolation, so really the more samples the better, however, diffuse surfaces generally don't have a high frequency, besides when a caustic forms, so the diffuse map doesn't need to be as high-resolution.

The Phong material is the material that is our diffuse surface, so it has an implementation of `compute_global_illumination()`, which inspects the photon maps, to retrieve colour information. This method uses the render settings to determine how many nearby photons to take from each map, and scales them according to the formula given by Jensen (1996). The formula essentially tells us that we taking the nearest n photons, and scaling them down based on the radius of the circle that the furthest point away creates. Technically we could use a sphere, however Jensen and Christensen (2000) tell us that in reality, most surfaces are relatively flat, so a flat circle is a better approximation.

Eventually this contribution will be summed with the contribution of the local lighting models, however there is a problem. We are using direct lighting calculations for local lighting, and a sampling technique, making many assumptions for our global illumination. We vary many parameters, so our lighting models don't nicely add together. The contribution of the photon maps needs to be scaled down.

This implementation uses the `NUMBER_OF_[type]_PHOTONS` constant to scale down by, as this is proportional to the area we expect to sample from on average (as the density will change proportionately). This means that if we scale in a certain way, we shouldn't have to worry the number of photons affecting our scales. After running the program a few more times, we converged on an appropriate scale to make the render look nice. We also need to pick an appropriate amount of points to sample from, Appendix **D.3** shows how these parameters can all change our renders. Appendix **D.4** shows a gallery of photon mapped scenes to see caustics in action.

4.3.1 Filtering

We can also apply 2D image filters to our photon map, once we've received the closest photons. This implementation includes the ability to turn on the cone filter, which essentially just says photons that are closer to our hit point are more important. In the renders, we can't see a dramatic difference when we apply the filter with varying `CONE_FILTER_CONSTANT` values (see Appendix **D.5**). We can even argue the caustics look more blurry with it (likely as our radius is below 1 anyway), however it's a useful setting for again fine tuning the image.

4.4 Stochastic Progressive Photon Mapping

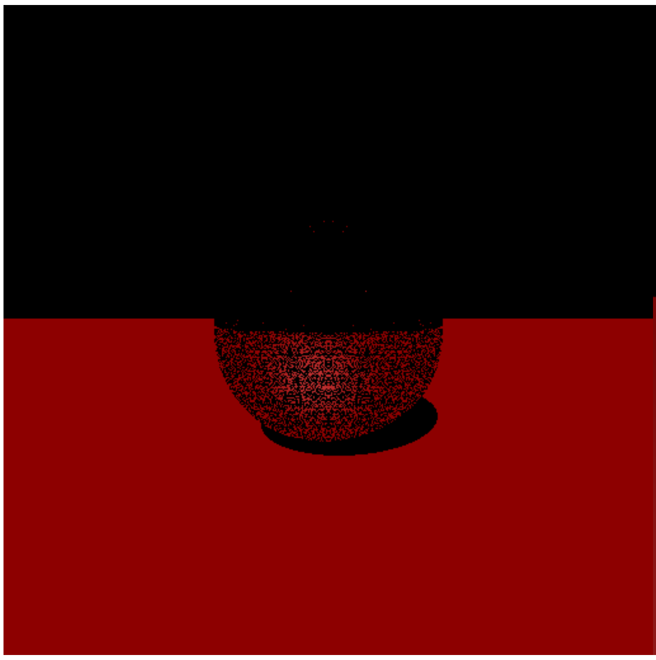
One problem with photon mapping is that it reduces noise compared to other methods, but it does this at the cost of slightly overblurring images without many photons. In theory you would need an infinite number of photons to converge to the true solution. Progressive Photon Mapping (Hachisuka, Ogaki and Jensen, 2008) was developed to help with this. It does the opposite to what we have implemented here, as it starts off with the raytracing pass, and then does numerous photon mapping passes, where the radius of the points sampled gets smaller and smaller with each pass. It starts with a weaker estimate of radiance (for us this is the energy or intensity), and adds more detail with every pass, reducing the memory footprint. This allows the render to eventually converge to the true value. It also allows users to see when the render is as accurate as they want, and stop whenever necessary without making guesses.

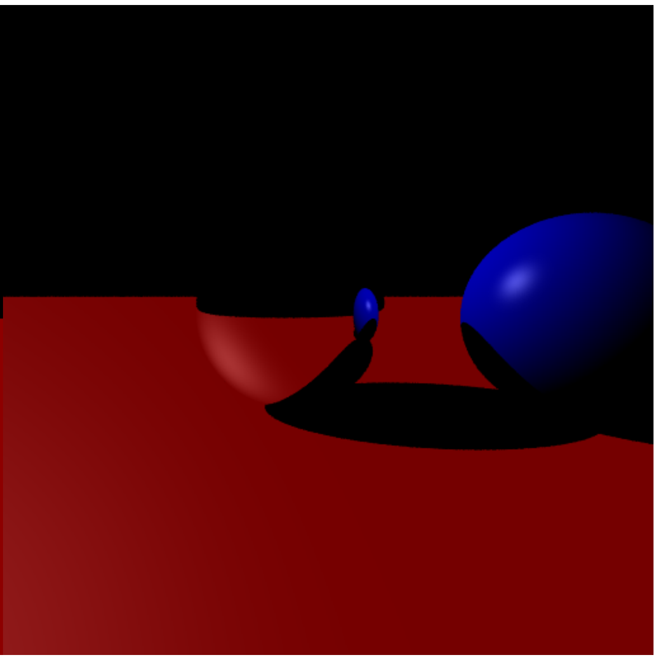
Progressive photon mapping struggled with sampling methods, which Hachisuka and Jensen (2009) call "distributed ray tracing". These effects include depth-of-field, motion blur, and glossy reflections. Hachisuka and Jensen (2009) then developed an algorithm known as Stochastic Progressive Photon Mapping (SPPM), which fixes this. SPPM looks at region convergence rather than points, and these can be sampled.

While for the sake of time these haven't been implemented in this raytracer, all the tools we would need are in the code base. We would only need a way to store the current estimate, and some logic for doing multiple photon mapping passes. Once that was in place, SPPM could be attempted by just sampling like we have after each photon mapping pass. If our program had a visual way to see the render progress, we could stop as soon as we were happy.

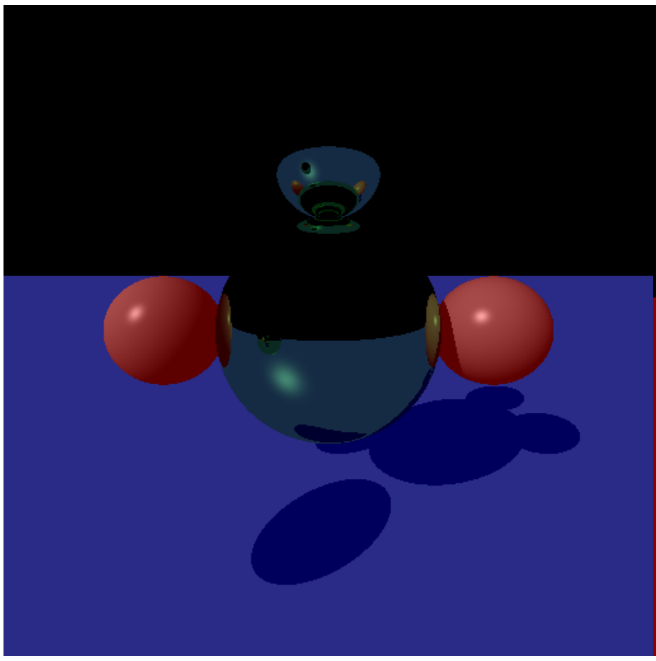
A Appendix: Global Lighting

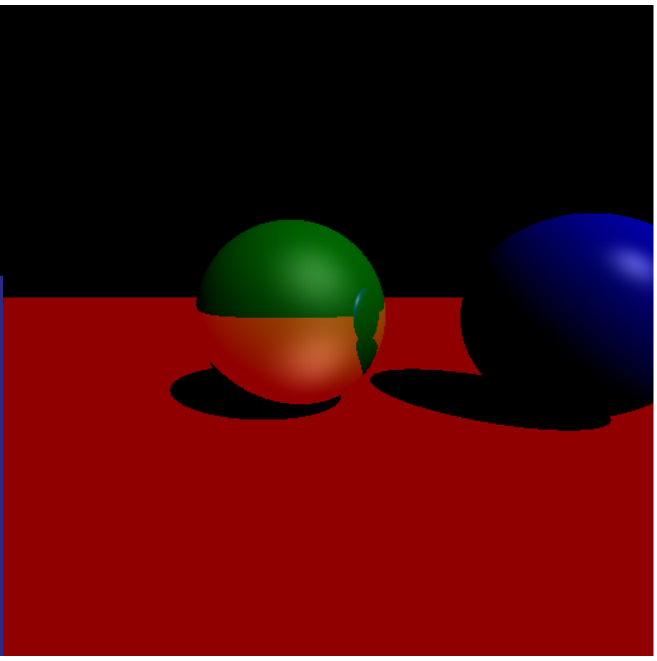
A.1 Spotty vs Clean Reflection


(a) Spotty reflection


(b) Reflection that isn't spotty

A.2 Reflection Stacking


(a) Reflection on multiple objects


(b) Reflection with a compound material

A.3 Refraction Gallery with Fresnel Terms

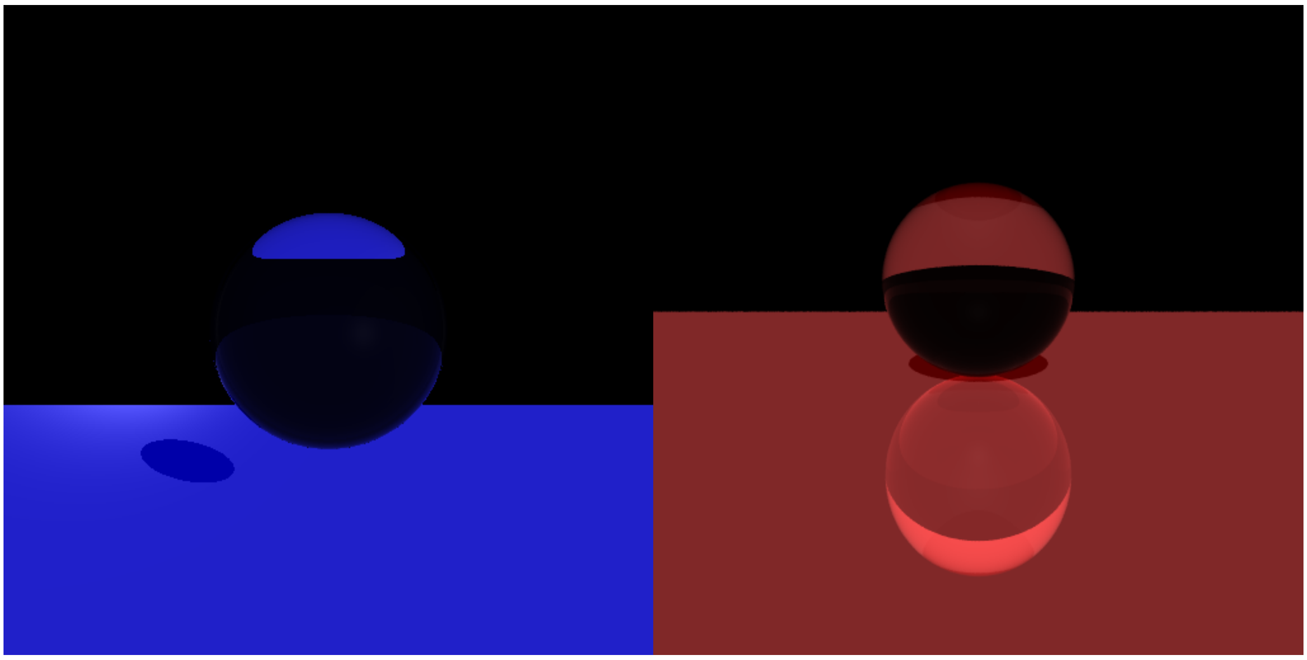


Figure 3: Renders showing the plane clearly reflected only near the tangent

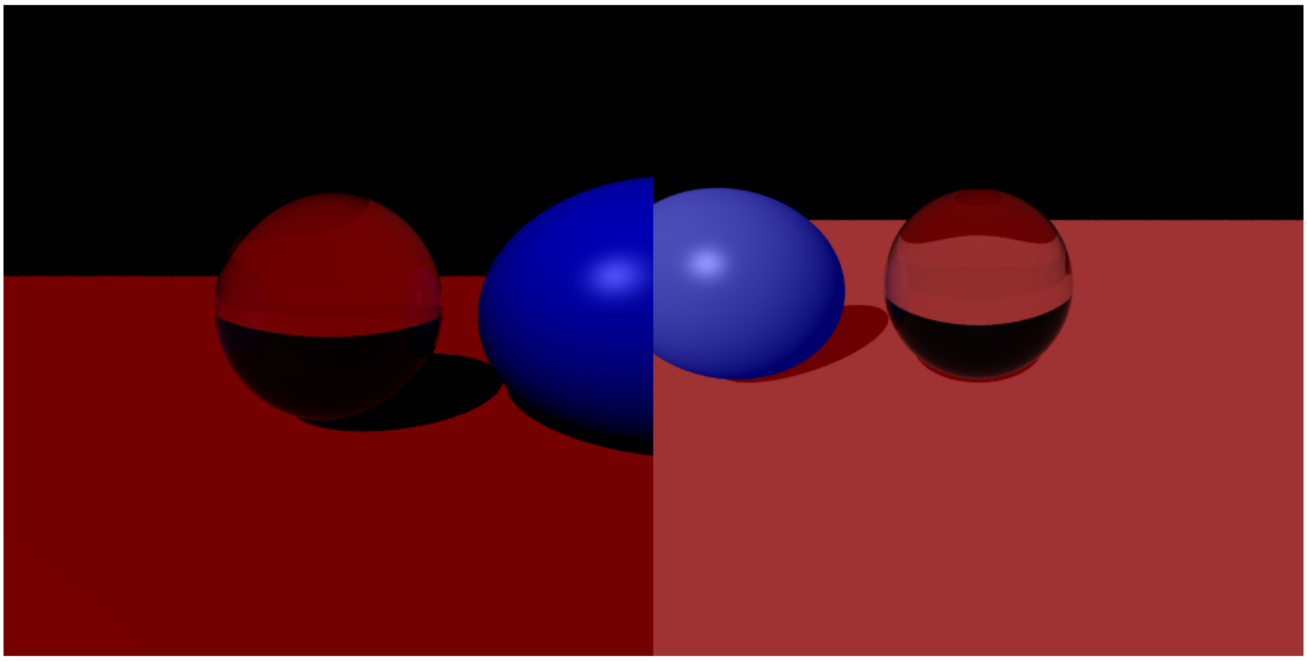
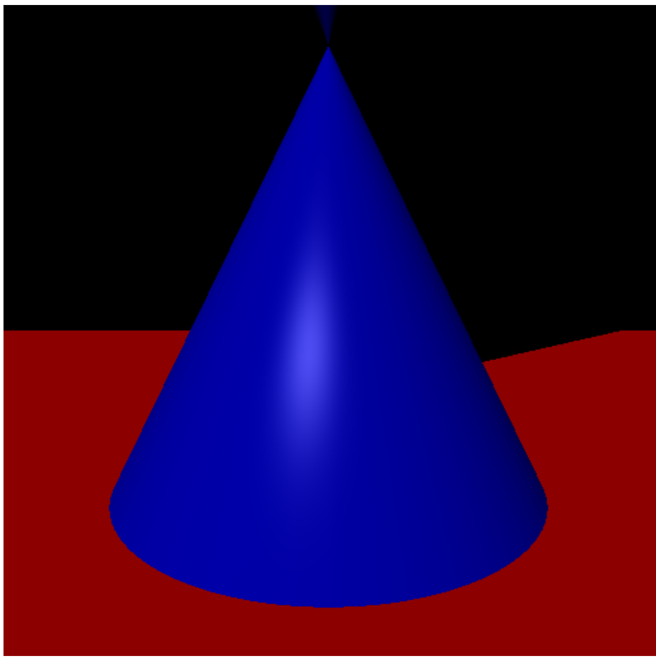


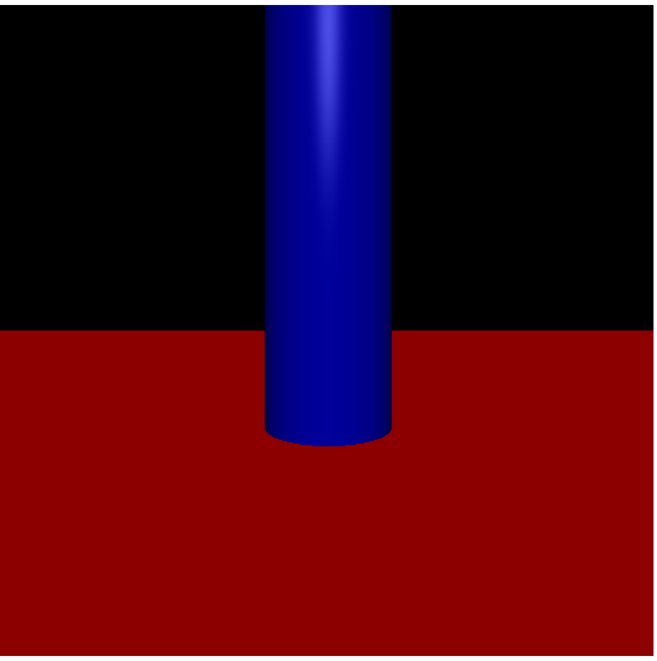
Figure 4: Renders showing the sphere being reflected on the sides

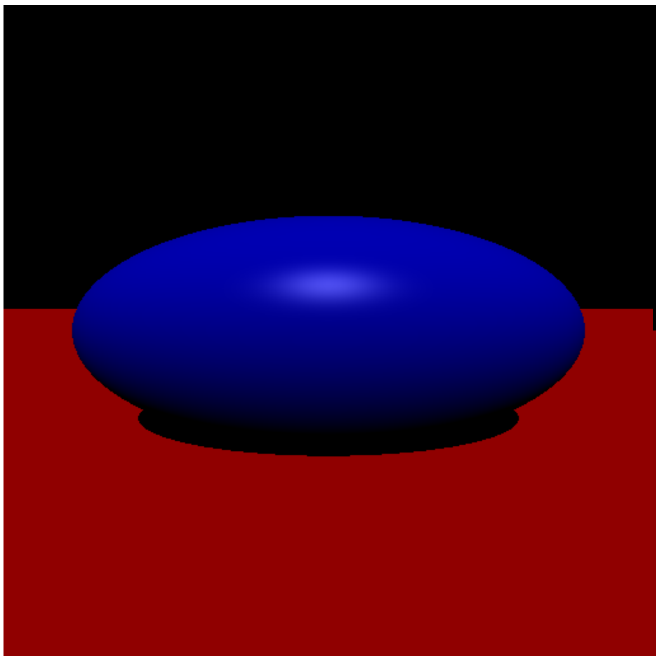
B Appendix: Quadratics

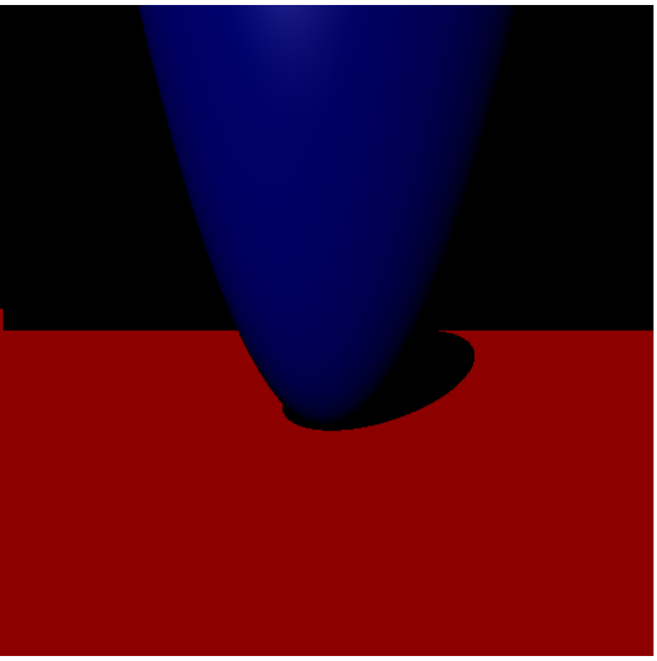
B.1 Quadratic Shapes

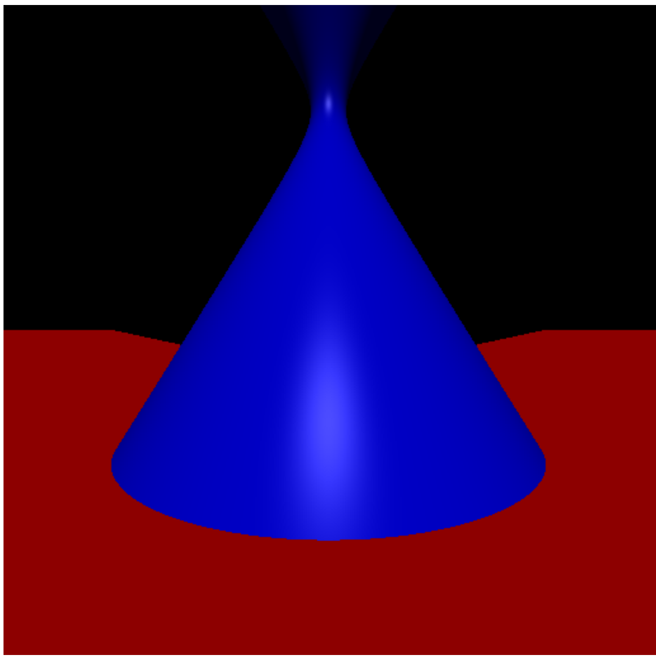
All quadratics are transformed for visibility.

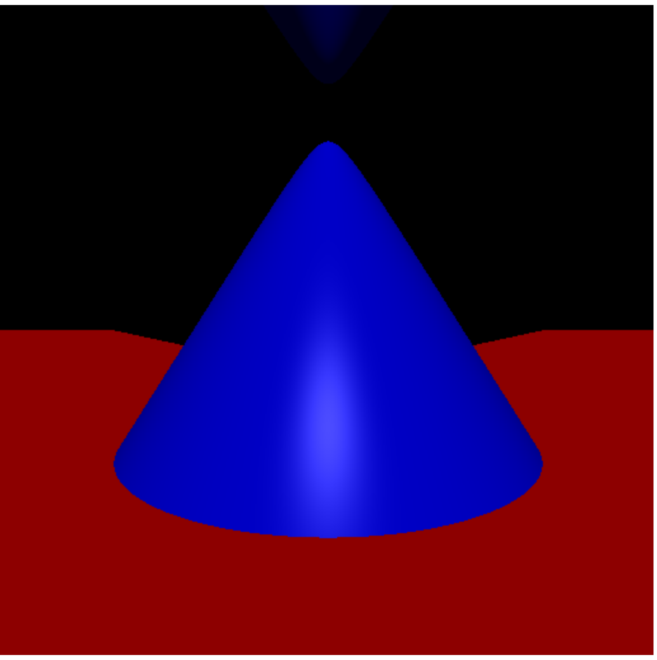

(a) Cone


(b) Cylinder

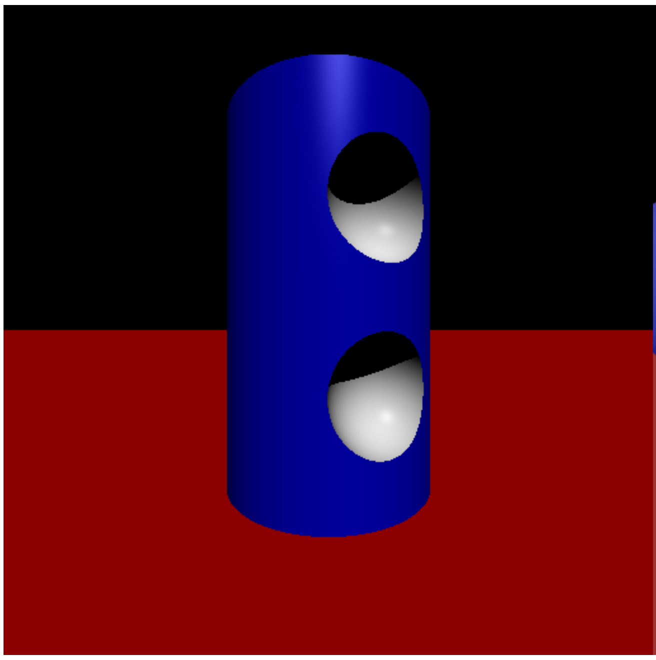

(a) Ellipsoid

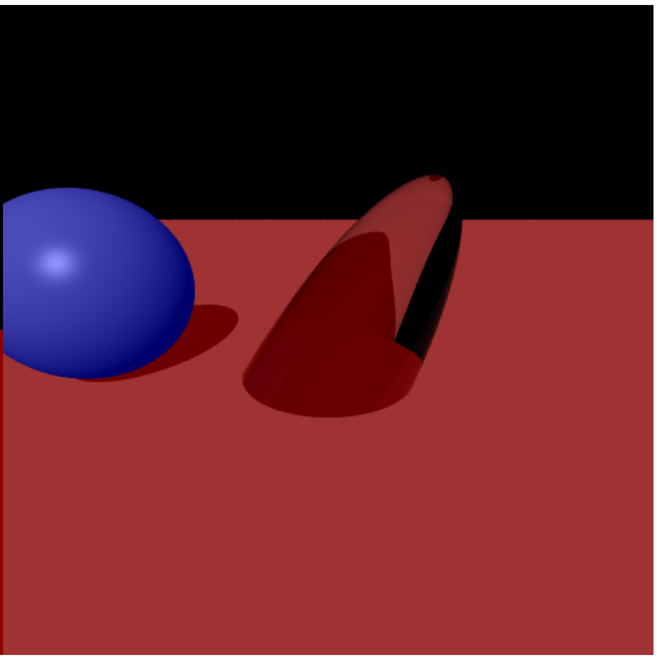

(b) Elliptic Paraboloid


(a) Hyperboloid of one sheet


(b) Hyperboloid of two sheets

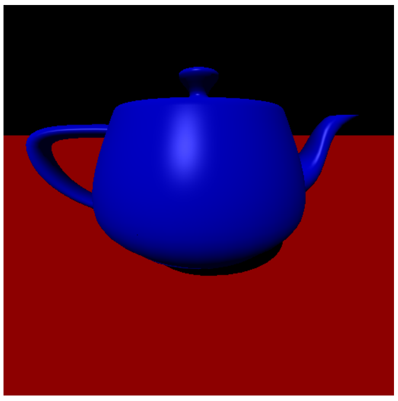
B.2 Fun with Quadratics

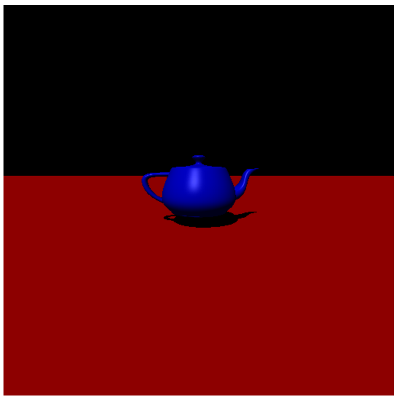

(a) Quadratic with CSG

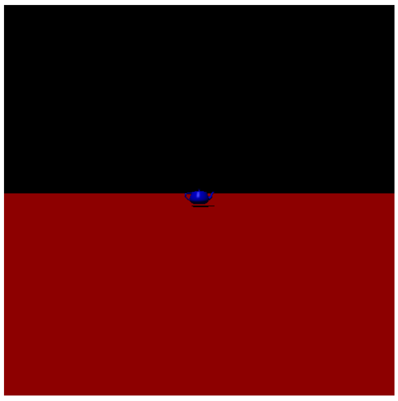

(b) Quadratic with glass

C Appendix: Acceleration

C.1 Distances


(a) Distance = 3


(b) Distance = 8


(c) Distance = 30

C.2 Statistics

Distance	Bounded	Seconds	Speedup
3	False	241	
3	True	241	1.12
8	False	225	
8	True	21	10.71
30	False	210	
30	True	1.9	110.53

Table 2: Bounding box speedup for a 512 × 512 pixel render

Distance	Threads	Seconds	Speedup
3	1	241	
3	8	67	3.59
30	1	210	
30	8	50	4.20

Table 3: Multithreading speedup for a 512 × 512 pixel render

Distance	Acceleration	Seconds	Speedup
8	Off	225	
8	On	8.46	26.60
30	Off	210	
30	On	0.74	283.78

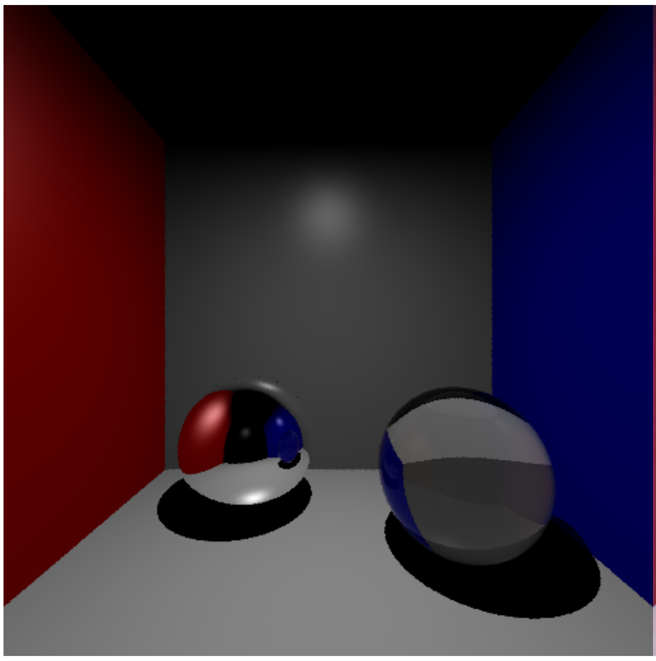
Table 4: Bounding box with parallelisation for a 512 × 512 pixel render

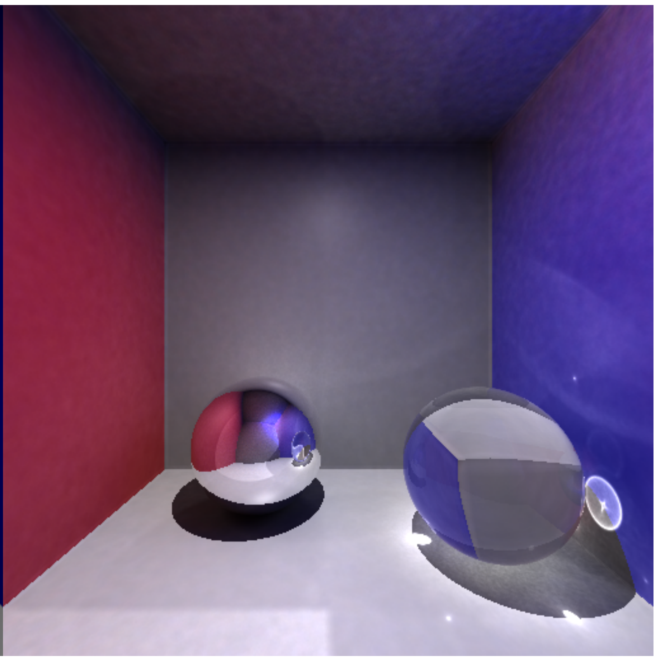
Multithreading	Seconds	Speedup
False	156	
True	31	5.03

Table 5: Multithreading on a photon mapped scene, for a 512 × 512 pixel **rendering pass**

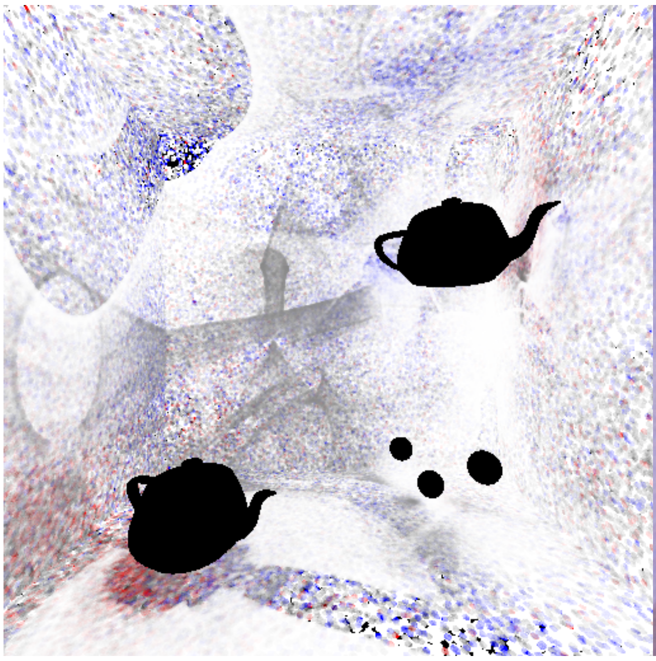
D Appendix: Photon Mapping

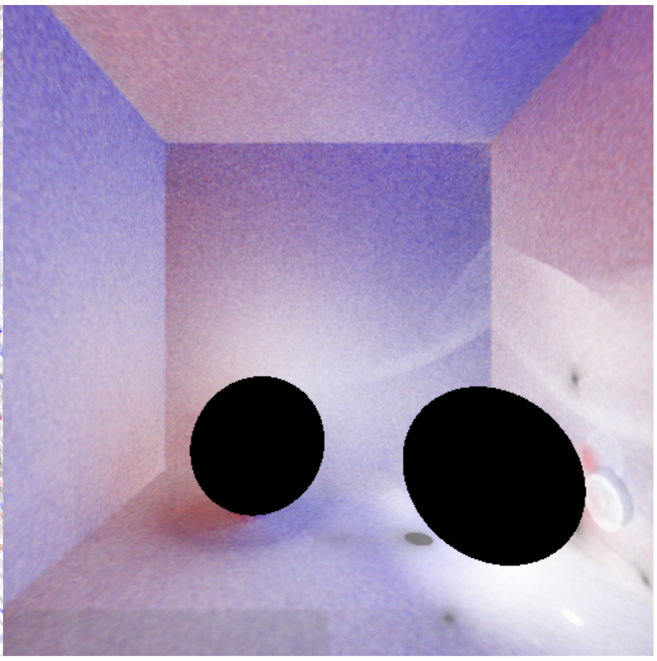
D.1 Global Illumination


(a) No Global Illumination (dark ceiling)

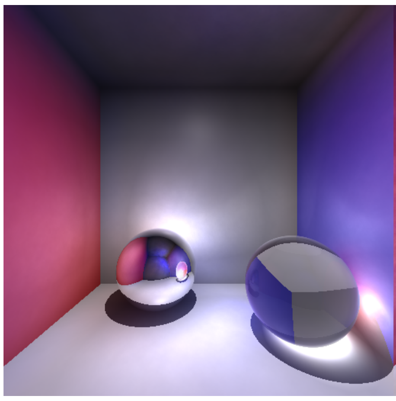

(b) Global Illumination

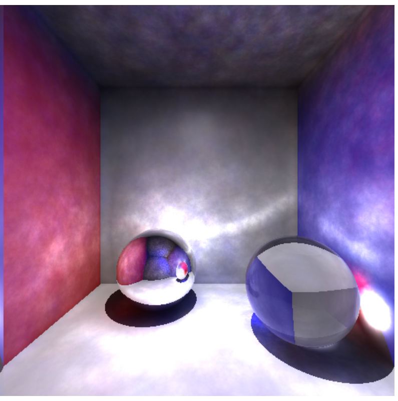
D.2 Photon Maps

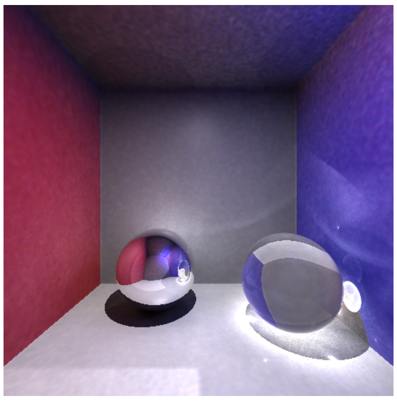

(a) Example caustic map


(b) High resolution photon map (both)

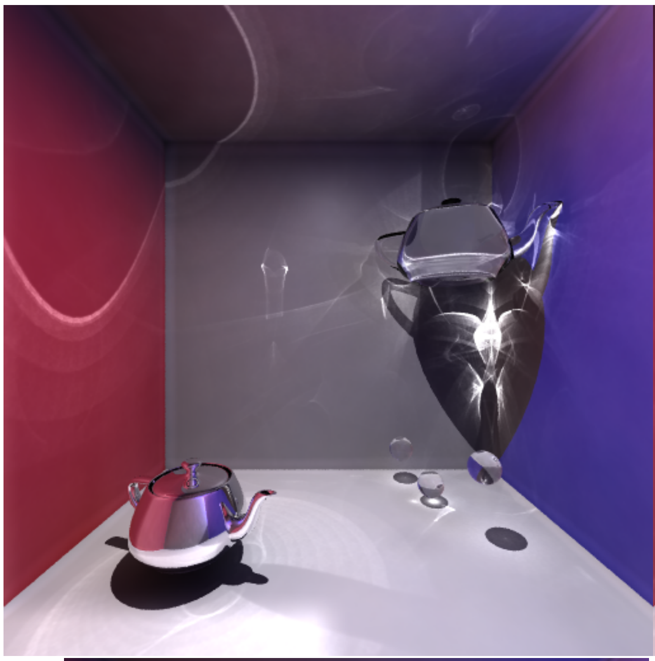
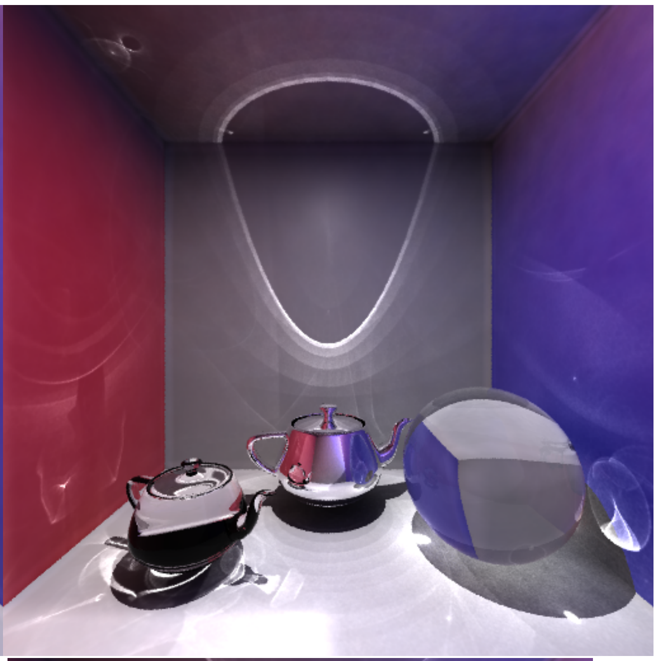
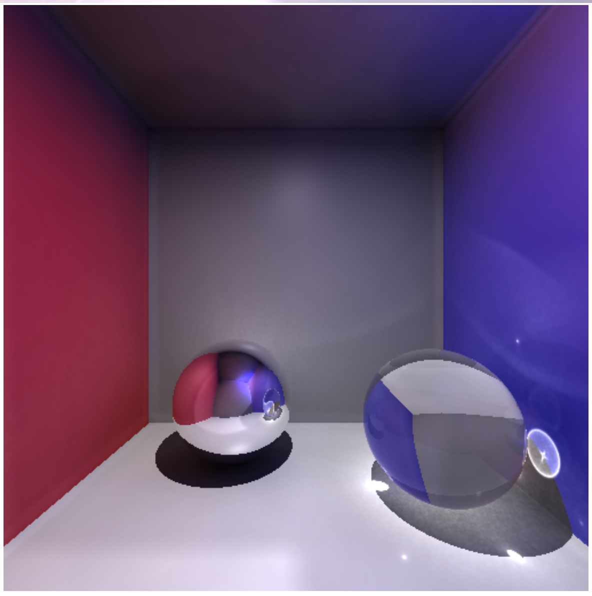
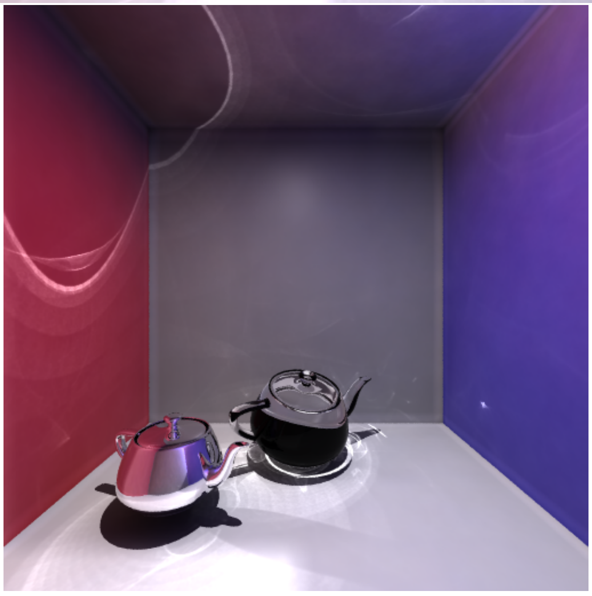
D.3 Bad Parameters


(a) Caustic map sampled too widely

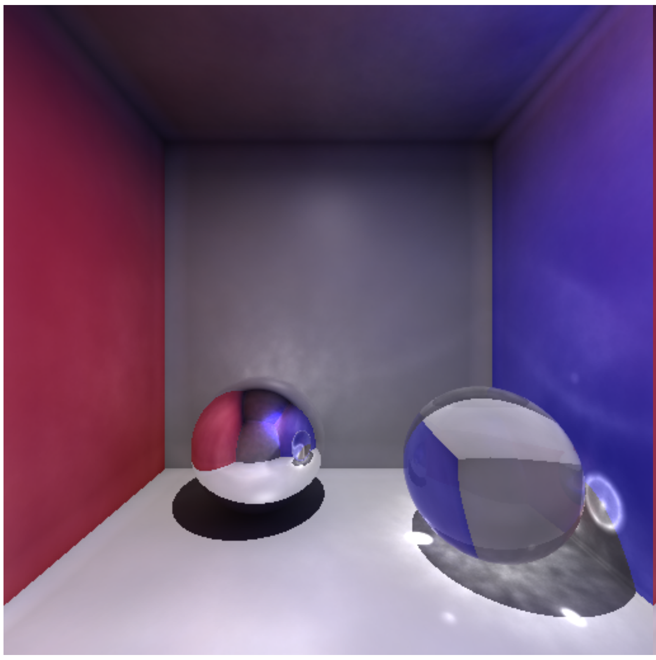

(b) Scaled too high

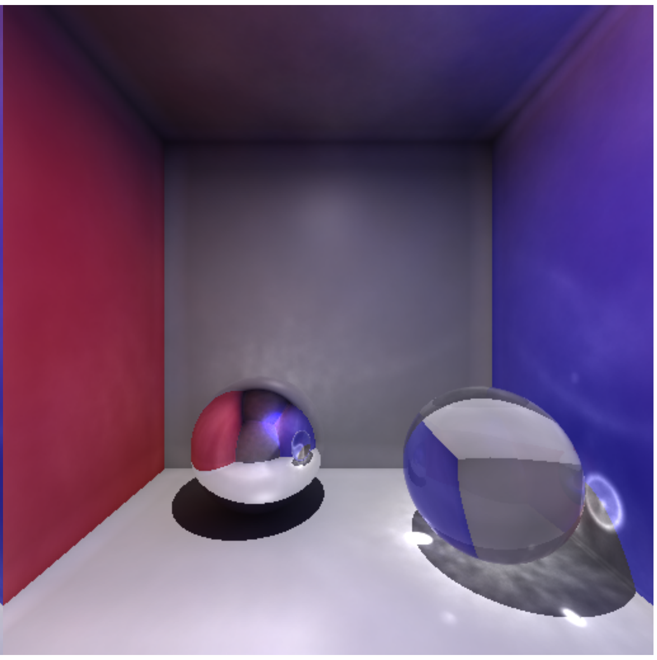

(c) Diffuse map sampled too sharply

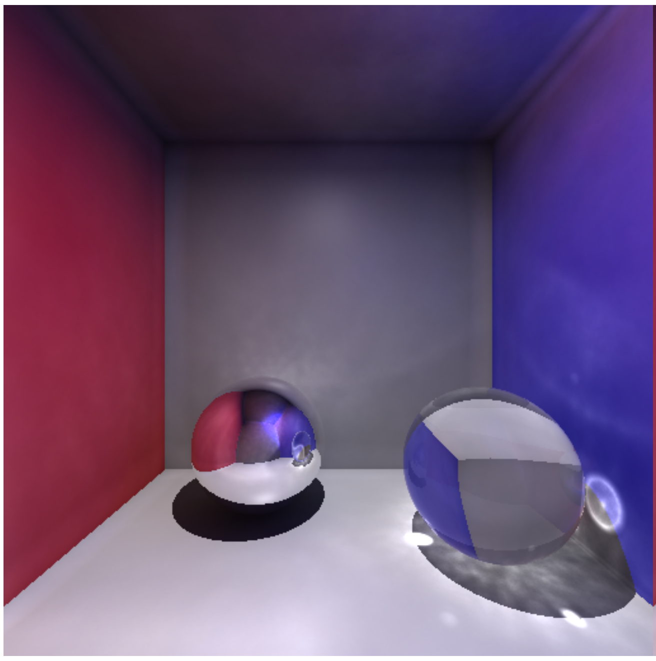
D.4 Photon Mapping Examples

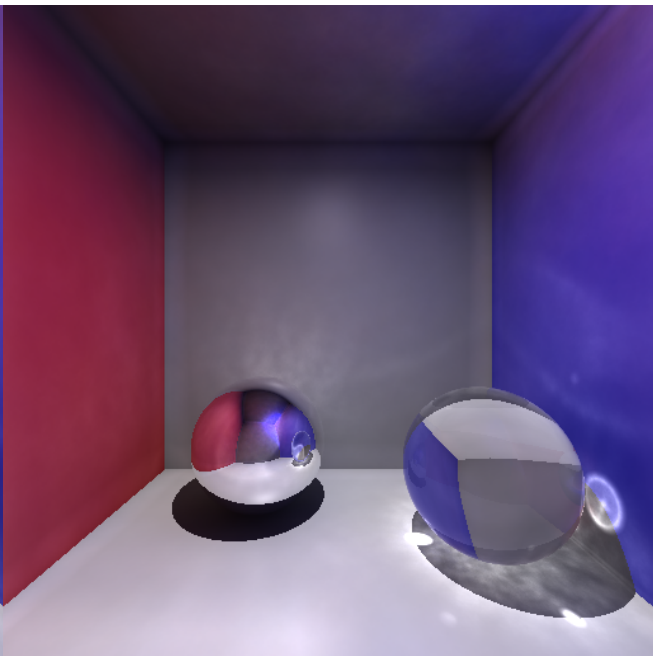





D.5 Filtering


(a) Cone constant off


(b) Cone constant = 8


(c) Cone constant = 20


(d) Cone constant = 40

References

Arm Developer, n.d. Embarrassingly parallel applications [Online]. Available from: `https://developer.arm.com/documentation/dui0538/f/parallel-processing-concepts/embarrassingly-parallel-applications`.

Arvo, J. and Kirk, D., 1990. Particle transport and image synthesis [Online]. *Proceedings of the 17th annual conference on computer graphics and interactive techniques*. New York, NY, USA: Association for Computing Machinery, SIGGRAPH '90, p.63–66. Available from: `https://doi.org/10.1145/97879.97886`.

Burk, K., 2017. *Kd::tree* [Online]. Available from: `https://github.com/xavierholt/kd`.

Cook, R.L., Porter, T. and Carpenter, L., 1984. Distributed ray tracing [Online]. *Proceedings of the 11th annual conference on computer graphics and interactive techniques*. New York, NY, USA: Association for Computing Machinery, SIGGRAPH '84, p.137–145. Available from: `https://doi.org/10.1145/800031.808590`.

Davis, D., n.d. *General quadric surface* [Online]. Available from: `https://www.geogebra.org/m/qAzgRphm#:~:text=The%20quadric%20surfaces%20equations%20can,equations%20in%20the%20new%20form`.

Goral, C., Torrance, K., Greenberg, D. and Battaile, B., 2002. Modelling the interaction of light between diffuse surfaces. *Computers graphics - cg*, 18.

Hachisuka, T. and Jensen, H.W., 2009. Stochastic progressive photon mapping [Online]. *Acm siggraph asia 2009 papers*. New York, NY, USA: Association for Computing Machinery, SIGGRAPH Asia '09. Available from: `https://doi.org/10.1145/1661412.1618487`.

Hachisuka, T., Ogaki, S. and Jensen, H.W., 2008. Progressive photon mapping. *Acm trans. graph.* [Online], 27(5). Available from: `https://doi.org/10.1145/1409060.1409083`.

House, D., n.d. *Quadric surfaces* [Online]. Available from: `https://people.computing.clemson.edu/~dhouse/courses/405/notes/quadrics.pdf`.

Jensen, H. and Christensen, N., 2000. A practical guide to global illumination using photon maps.

Jensen, H.W., 1996. Global illumination using photon maps. *Proceedings of the eurographics workshop on rendering techniques '96*. Berlin, Heidelberg: Springer-Verlag, p.21–30.

Jensen, H.W., 2001. *Realistic image synthesis using photon mapping*. AK Peters.

Lafortune, E. and Willems, Y., 1998. Bi-directional path tracing. *Proceedings of third international conference on computational graphics and visualization techniques (compugraphics'*, 93.

Mebius, J.E., 2007. Derivation of the euler-rodrigues formula for three-dimensional rotations from the general formula for four-dimensional rotations. `math/0701759`.

Peter Shirley, Trevor David Black, S.H., 2023. *Ray tracing in one weekend* [Online]. Available from: `https://raytracing.github.io/books/RayTracingInOneWeekend.html`.

Royal Meteorological Society, 2020. How are rainbows formed? [Online]. Available from: `https://www.rmets.org/metmatters/how-are-rainbows-formed#:~:text=Rainbows%20are%20formed%20when%20light,air%2C%20such%20as%20a%20raindrop`.

Solomon, J., 2020a. *Accelerated ray tracing; bounding volumnes; kd trees* [Online]. Available from: `https://youtu.be/TrqK-atFfWY`.

Solomon, J., 2020b. *Ray tracing; refleciton and refraction, ray trees* [Online]. Available from: `https://youtu.be/Tyg02tN9oSo`.

Veach, E. and Guibas, L.J., 1997. Metropolis light transport [Online]. *Proceedings of the 24th annual conference on computer graphics and interactive techniques*. USA: ACM Press/Addison-Wesley Publishing Co., SIGGRAPH '97, p.65–76. Available from: `https://doi.org/10.1145/258734.258775`.