Advanced Computer Graphics: Raycaster with Photon Mapping and Depth of Field

Jac Griffiths

December 16, 2021

1 Objective

I aim to program a ray caster enhanced by photon maps which are constructed as a pre-processing step, capable of rendering the following effects:

- Local lighting calculations using a BRDF
- Full specular reflection
- Refraction of light through objects with an arbitrary index of reflection
- Diffuse interreflections
- Caustics on diffuse objects caused by refraction and reflection of light
- Depth of field effect using sampling

2 Ray Tracing

2.1 Specular Reflection

In order to calculate the trajectory of the reflected ray, I used:

$$\vec{R} = \vec{I} - 2(I.N)\vec{N} \tag{1}$$

(where R = reflected direction vector, I = incident direction, N = normal)

This calculation was already being done as part of the Phong local lighting calculations, so I abstracted it in the physics.cpp file for more general use.

Contributions from reflections are calculated during the rendering step. View rays fired from the camera are reflected from the surface, creating secondary rays that recursively apply the rendering step and contribute to the radiance returned, weighted by the surface's reflection coefficients. This induces a tree of view rays for each primary ray fired. To ensure that this doesn't recurse indefinitely, and to improve efficiency, a 'depth' parameter is used to control the maximum depth of this tree. Also, in order to optimise, the surface's reflection coefficient is checked to be above a certain small threshold before the reflection ray is created and traced, to ensure that it's contribution won't be negligible.

2.2 Refraction

Similarly to specular reflection, refraction is also accounted for during the rendering step. If an appropriately transparent surface is hit by a primary view ray, the rendering step is applied recursively to rays that are refracted according to the surface's properties (normal and index of refraction).

To calculate the trajectory of the refracted ray, Snell's law is used. The direction vector of the incident light is inverted, and then used in a dot product with the normal of the surface to calculate the cosine of θ_i. This value ends up being negative if and only if the smallest angle between I and N is greater than 90°- when the normal is pointing to the other side of the surface to I. In these cases, my code will temporarily invert the normal - this means that all surfaces are two-sided when it comes to refraction.

Next, $\cos(\theta_t)$ can be calculated using Snell's Law, and the trajectory of the refracted ray is given by (using CM30075 lecture slides):

$$T = \frac{1}{\eta}\vec{I} - (\cos(\theta_t) - \frac{1}{\eta}\cos(\theta_i))\vec{N} \tag{2}$$

The proportion of light refracted and reflected should not only be dependent on the coefficients of the material, but also on the viewing angle (see Figure 1 below). The refraction function takes an additional two pointer arguments to which the results of the Fresnel Equations are written. These two results are then used to weigh the effect of reflected and refracted rays in the tree. Because the Fresnel coefficient is required for the reflection step, refraction must come first in the ray tracing code such that the coefficient can be calculated.

If the light is travelling from a material with a higher index of refraction to a lower one, then it may be the case that θ_i is greater than the critical angle for the surface, causing all light to be reflected and none to be refracted. This is known as total internal reflection. The critical angle can be found by applying Snell's law for $\theta_t = \pi/2$, giving $sin(\theta_c) = \frac{\eta_2}{\eta_1}$. Notice that when $\theta_c < \theta_i < \pi$:

$$sin(\theta_i) > \frac{\eta_2}{\eta_1} \Rightarrow \frac{\eta_1^2}{\eta_2^2}\sin^2(\theta_i) > 1 \Rightarrow \cos^2(\theta_t)^2 = 1 - \frac{\eta_1^2}{\eta_2^2}\sin^2(\theta_i)^2 < 0$$

Therefore, it is possible to check for total internal reflection by checking if $\cos^2(\theta_t) < 0$ before taking its square root. In this case, the refraction is skipped, and the Fresnel term for reflection is set to 1.

One problem that arises is getting the correct indices of refraction for the medium the ray was previously inside, and the one it is about to enter. I solved this by storing the object the ray is inside as an argument of the raycast function. If the ray is intersecting the same object it is currently inside, it is assumed to be leaving the object. A limitation of this solution is that it may incorrectly handle a scene with intersecting transparent objects, however this shouldn't happen in the first place so I believe an extended solution that uses a stack of some kind would be unnecessary.

2.3 BRDFs

In order to implement photon mapping, I needed to change my code to support general BRDFs. I created a new abstract class for BRDFs, with the abstract function getReflectedLight() which uses the incident direction, reflect direction and normal direction to calculate the proportion of light in each colour channel that is reflected in the reflect direction. BRDFs can be used more compactly with only 4 floating point arguments (two angles for each of the incident and reflect vectors), however this interface proved more useful and readable for my code as vectors of the required form were already available in the sections of code that would use the BRDF.

I implemented the Phong lighting model as a subclass of the BRDF abstract class, and attached it to each of the objects in the scene. Then, I replaced the local lighting calculations in the raytracing function with a call to the BRDF getReflectedLight function. As I currently only have the Phong BRDF implemented, the functionality hasn't changed, but this structure allows for applying arbitrary BRDFs to different objects in the scene, changing both the local lighting calculations and the behaviour of photons during the photon mapping step.

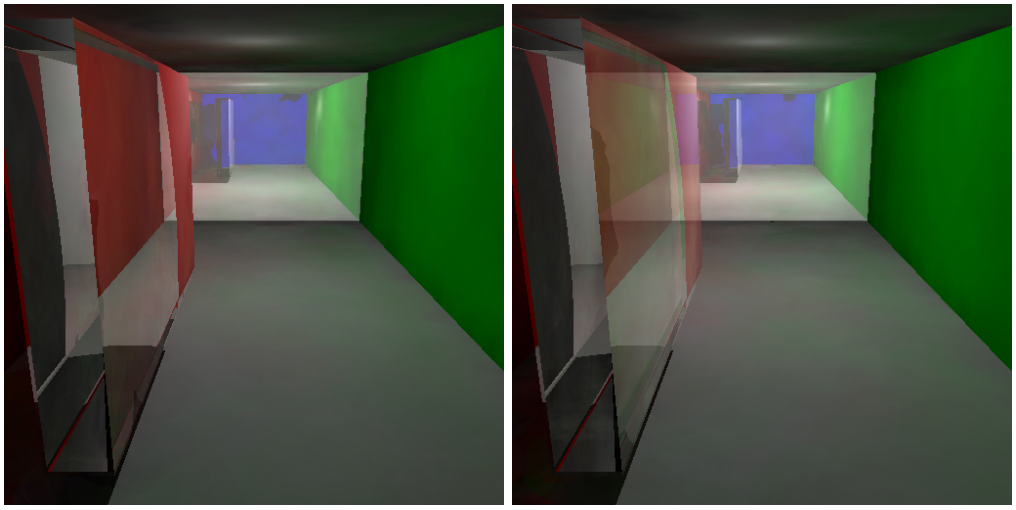


Figure 1: A transparent cuboid. Left: Without weighing by Fresnel terms, Right: With Fresnel terms. In both pictures, the left wall is red and the right is green, but only the second picture shows some reflection from the right wall. In both cases, you can see total internal reflection on the closest face of the cuboid.

3 Photon Mapping

3.1 KD Tree

Before generating photons, I needed a way to store them along with their position, such that photons near a certain point can be efficiently retrieved. As recommended by Jensen, I decided to use a KD-tree for this purpose, as it allows retrieval of M photons in a tree with N photons in $O(Mlog_2(N))$ time [Jensen, 1996]. A KD-tree works similarly to a binary tree, except that it is sorted based on multiple values (in this case 3: x, y and z), and the value used to order nodes depends on the depth in the tree. This means that some extra checks are required while traversing the tree compared to a regular binary tree before cutting a branch.

3.2 Generating Photons

Currently, my code only supports photon mapping from point lights. In the createGlobalPhotons function, rays are created at the position of the point light, and fired in random directions. Then, each ray performs an intersection test, creating a photon at the intersection position and storing it in the KD Tree. A Boolean 'direct' argument is used by this function to decide whether to store as a direct photon or an indirect one; the primary rays call the function with direct set as true, while any recursive calls have it set to false.

Next, the russianRoulette() function uses the coefficients in the hit surface's material as a probability to decide whether the photon is absorbed or reflected according to the BRDF (specular reflections and refractions are considered separately when creating photons for caustics). If the latter is chosen, a new random direction is selected, and a ray is recursively fired in that direction with power determined by the BRDF. This recursion will terminate when one of four conditions are met:

- The total power of a photon is less than a certain low, fixed threshold (so contribution is negligible)
- A ray does not intersect with any objects in the scene
- The russianRoulette() function determines randomly that the photon should be absorbed
- The recursion tree reaches a set maximum depth

In order for accurate caustics to be produced, a larger number of photons need to be fired at specular objects. Each object in the scene is assigned a bounding sphere when the scene is created. The fireRaysTowardsSpecular() function finds a random point within a given specular object's bounding sphere, then the direction vector for a photon can be set to the unit vector in the direction going to this position from the point light.

One problem I ran into while coding this was that a sphere, as a subclass of object, cannot be stored as an attribute within the object class. I solved this by making the pointer to the bounding sphere an object pointer, and then dynamically casting to a sphere pointer at the start of the function. If this cast is unsuccessful, an error message is printed, and no photon rays can be fired at the object.

3.3 The Rendering Equation

Kajiya presents a generalised solution known as the rendering equation [Kajiya, 1986]. Within the context of a raycaster, each ray should retrieve the amount of radiance (in each colour band) moving towards the eye from the point of intersection of the ray with the scene, which can be defined as $L(x', \omega')$ where x' is the intersection point, and ω' is the ray's direction (from the intersection back to the eye). The rendering equation is given by:

$$L(x', \omega') = E(x', \omega') + \int \rho_{x'}(\omega, \omega') L(x, \omega) G(x, x') V(x, x') dA \tag{3}$$

The raycaster handles the visibility term, $V(x, x')$ by finding the first intersection, and the geometric relationship term G(x, x') is handled by the BRDF. I also do not plan to add emissive surfaces to by code, and so $E(x', \omega')$ can be ignored. Thus, for each view ray, I will consider the integral:

$$L(x', \omega') = \int \rho_{x'}(\omega, \omega') L(x, \omega) dA \tag{4}$$

This equates to continuously summing the contents of the integral over light coming from each possible incoming angle. I could take discrete samples of incoming light from various angles using the photon map, apply the BRDF and sum them - but doing this directly for each radiance value needed will be far too expensive. Instead, following the method presented by Jensen [Jensen, 1996] [Jensen and Christensen, 2000], the integral can be separated according to different light paths, and different methods can be used to approximate each of them separately. The categories considered are:

- Any Specular Reflection: L(D/S)*SE
- Direct light reflected diffusely: LDE
- Diffuse interreflection: LD*DE
- Caustics: LS*DE

The first category is handled in the rendering step by casting secondary view rays (see sections 2.1 and 2.2), and the second is handled by local lighting calculations using the BRDF (however photon maps can be used to optimise - I will cover this in section 3.4). It remains to add the contributions from the other two, which I will cover in sections 3.5 and 3.6.

3.4 Optimising Direct Illumination

The shadow checks made during local lighting calculation can be optimised by generating 'shadow photons' during the photon mapping step. Whenever a direct photon makes context with a surface, the program will repeatedly create rays on the same trajectory as the direct photon, perform intersections tests and record

shadow photons in areas which direct light does not reach. Then, during the rendering step, most shadow tests can be completed much more quickly by comparing the amount of shadow photons and direct photons in a small area around the point of intersection. As many scenes have large areas that are either in complete shadow or no shadow at all, this shortcut applies in a lot of cases. However if the number of direct versus shadow photons is not decisive, a shadow ray should still be cast, as these are most likely to be areas between shadows and light, where it is important that the outline of the shape of the shadow is accurate.

In my implementation, I wrote a function 'shadowChance' which returns the number of shadow photons in a small area divided by the total of shadow and direct photons in that same area. If no photons are found, no conclusions can be made about whether the area is in shadow or not, so the function returns 0.5 (although this case shouldn't happen often as long as enough global photons are fired). This is called by the renderer each time a shadow check needs to be made, and as long as the chance is decisive (less than 0.1 or greater than 0.96), the shadow ray step is skipped.

As intersection tests are still performed in the photon mapping step, this approach scales just as poorly with scene size, however it doesn't get slower when depth or image resolution are increased. Additionally, if the photon mapping step can be skipped (i.e. lighting has been 'pre-baked') then most intersection tests are cut out entirely. This would be especially useful in an animation or video game, where the camera is moved around the scene and the rendering step needs to be repeated, but the photon mapping step does not (as long as the lights are stationary).

3.5 Diffuse Interreflection

To get more realistic global illumination, the global indirect photon map is used, as it contains photons from rays that are reflected according to the BRDFs of surfaces in the scene. At the point x', the KD tree is searched for all photons within a radius of 0.5, and the code calculates the average of their direction vectors. This way, the colour-bleeding effects are a lot smoother than they would be if only the closest photon was used - I needed to offset the fact that photon maps are a discrete estimate of how light bounces around the scene. The average power of the photons in each colour channel is also calculated, and used along with the average direction and the BRDF of the surface to calculate the radiance in the direction ω'.

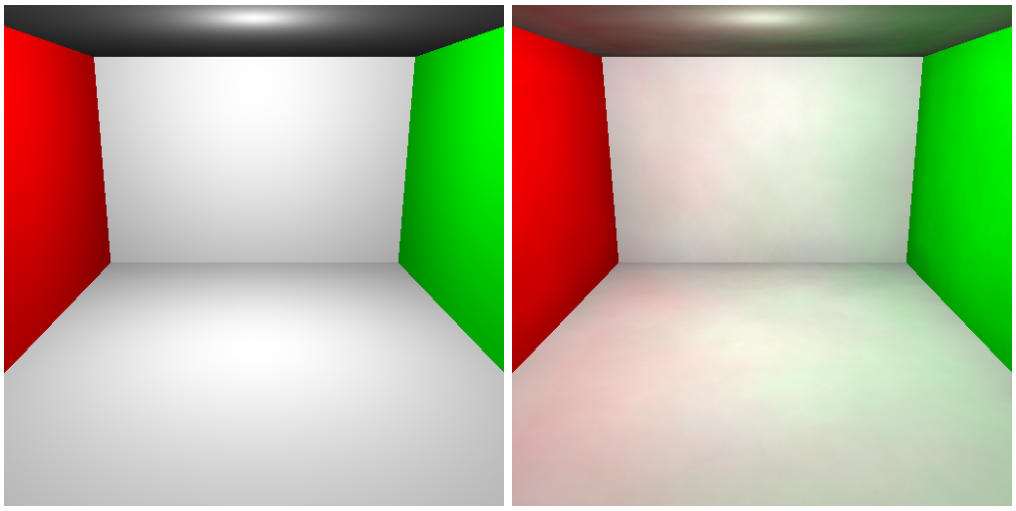


Figure 2: Left: 0 global photons fired (so no ambient light). Right: 15000 global photons fired

3.6 Caustics

As the outlines of caustics should be rendered accurately, a much smaller area around x' should be considered when searching for photons - I found that a sphere of radius 0.1 gives good results. Apart from this,

the algorithm for rendering caustics is similar to the one for diffuse interreflection - with one important distinction.

While rendering the diffuse interreflection, the random direction of photons from the point light, amount of photons and nature of diffuse reflection allowed the assumption that photons will be spread evenly throughout the scene. However, as caustic photons are targeted at specular objects, their density within the area searched is important. There will be many areas with a relatively small amount of photons, and these areas should appear dimmer, so simply averaging the brightness of photons in the area will be inaccurate. See the image produced below, where the bright circles on the walls are incorrectly rendered at a similar brightness level to the focused light below the sphere, despite there being very few caustic photons landing on the walls compared to the many that land below the sphere. Additionally, the caustic pattern below the sphere is incorrect around the edges.

Instead, the brightness at a point should be reliant on the amount of photons found in the search, so I needed to divide the sum of the power values of nearby caustic photons by the total amount of caustic photons in the scene. However, as there are many caustic photons, this value needs to be upscaled by some value that represents the highest proportion of caustic photons contained within one sphere of radius 0.1 within the scene. Calculating this value would require a pre-processing step - instead, I have opted to allow the user to manually set this scaling value in the code.

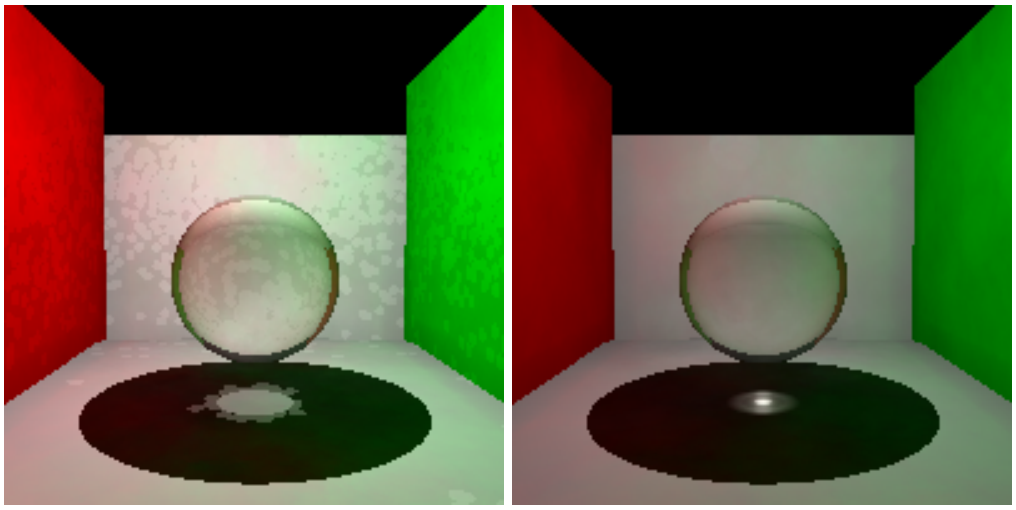


Figure 3: Left: The same algorithm for diffuse interreflection being incorrectly applied to caustics. Right: The improved algorithm.

4 Additional Features

4.1 Depth of Field

My implementation of the depth of field effect uses the small lens model (in particular, the approach outlined in the CM30075 lecture slides). After casting the usual primary ray for a pinhole camera model, a point is selected randomly in a disc around the pinhole (where the disk is parallel to the image plane) which I will call the lens. Consider a 'focal plane' , parallel to the image plane at certain distance perpendicular to the camera, called the focal distance. Another primary view ray is created that starts at the new point on the lens, and moves in a direction such that it intersects the original primary ray on the focal plane, as shown in Figure 4 below.

The aim is to scale the direction vector of the original ray such that it ends on the focal plane ($\vec{p}$), such that it could be added to the vector going from the new ray's start position to the pinhole ($\vec{d}$), and get

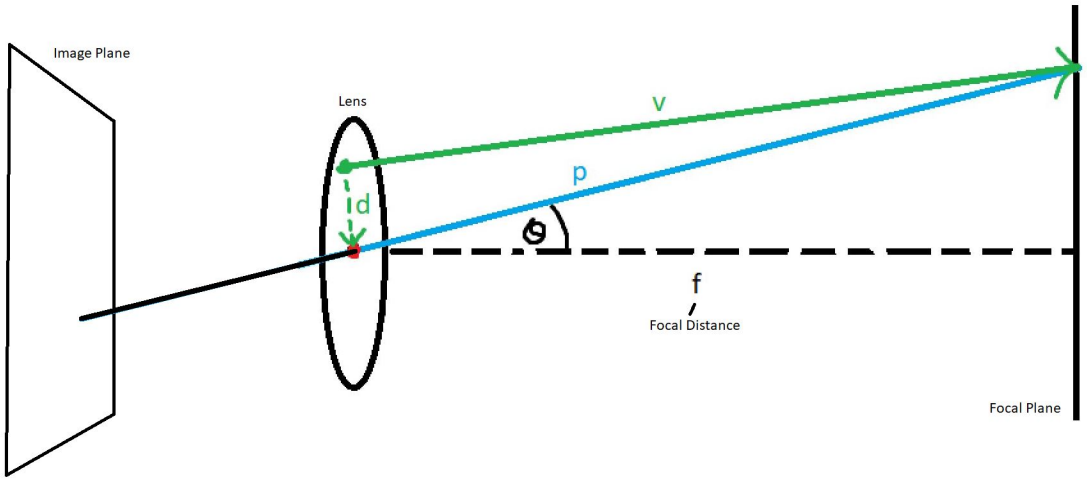


Figure 4: The small lens model used for depth of field effect.

an scaled version of the new direction vector that ends on the focal plane ($\vec{v}$). Normalising this gives the direction vector for the new ray. See in the image that $\vec{v} = \vec{d} + \vec{p}$, and that a normalised version of v is the direction vector needed for the new ray.

Therefore, it remains to figure out how to scale the direction vector of the original primary ray such that it ends on the focal plane (as $\vec{p}$ does in the image). Notice that the focal length, length of $\vec{p}$ and focal plane form a right-angled triangle with p as the hypotenuse, and so:

$$|\vec{p}| = \frac{f}{\cos(\theta)} \tag{5}$$

where $\cos(\theta)$ can be calculated by taking the dot product of the original direction vector and the vector pointing perpendicular to the lens (which is given by one of the orthonormal basis vectors of the pinhole camera).

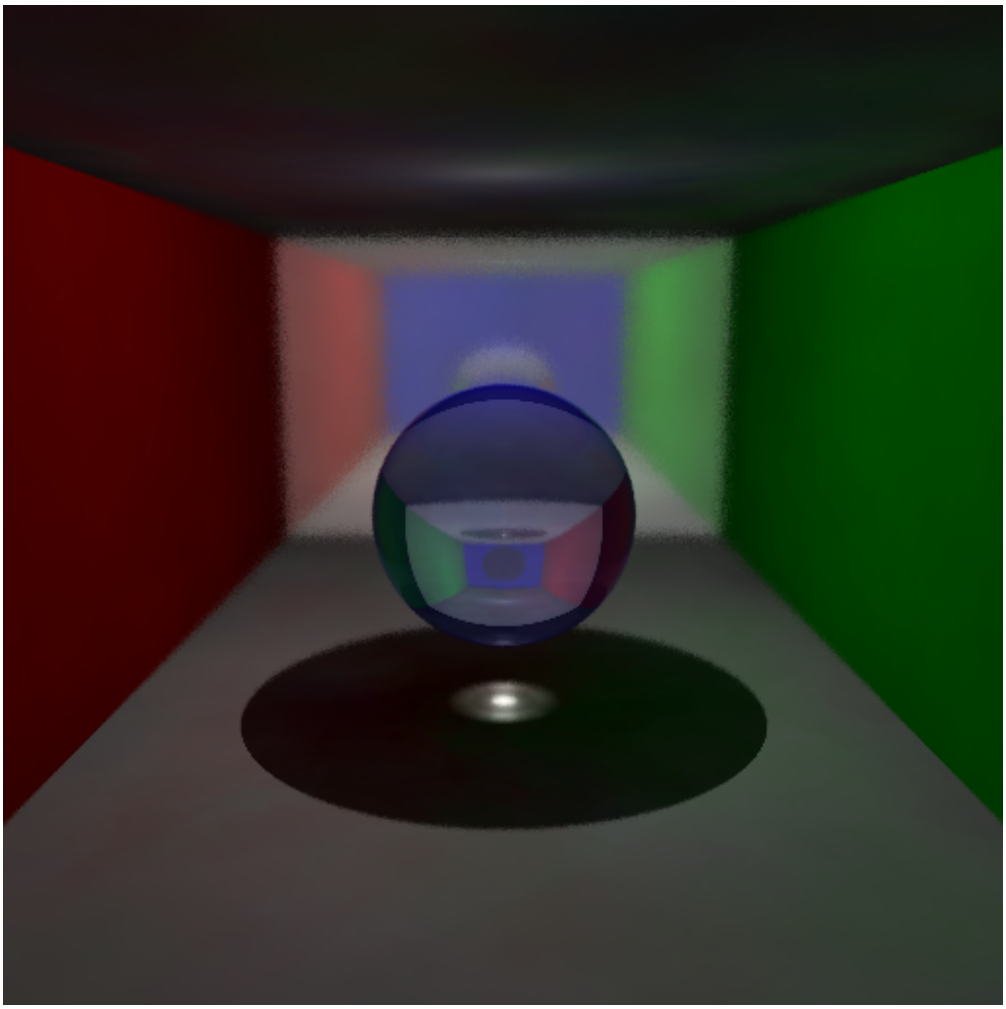
The makeSampledRays() method in the pinholecamera.cpp file generates rays that start at random positions on the lens, and meet the original ray at the focal plane using the method described above. In the rendering step, rather than one ray being created for each pixel in the framebuffer, a given number of samples are created in this way, and the average of their raytracing results gives the final colour of the pixel.

4.2 Optimising with Bounding Spheres

The bounding spheres from section 3.6 can also be used to optimise intersection tests for polymesh objects in the scene. Before performing all the triangle intersections, one additional intersection test with the bounding sphere is performed. If this fails, all triangle intersections can be skipped. This ends up being far more efficient, as many rays will not intersect a given mesh, and therefore will be able to skip the triangle intersections. In order for the polymesh to be rendered properly, it should be ensured that the bounding sphere encompasses all of its points.

5 Appendix

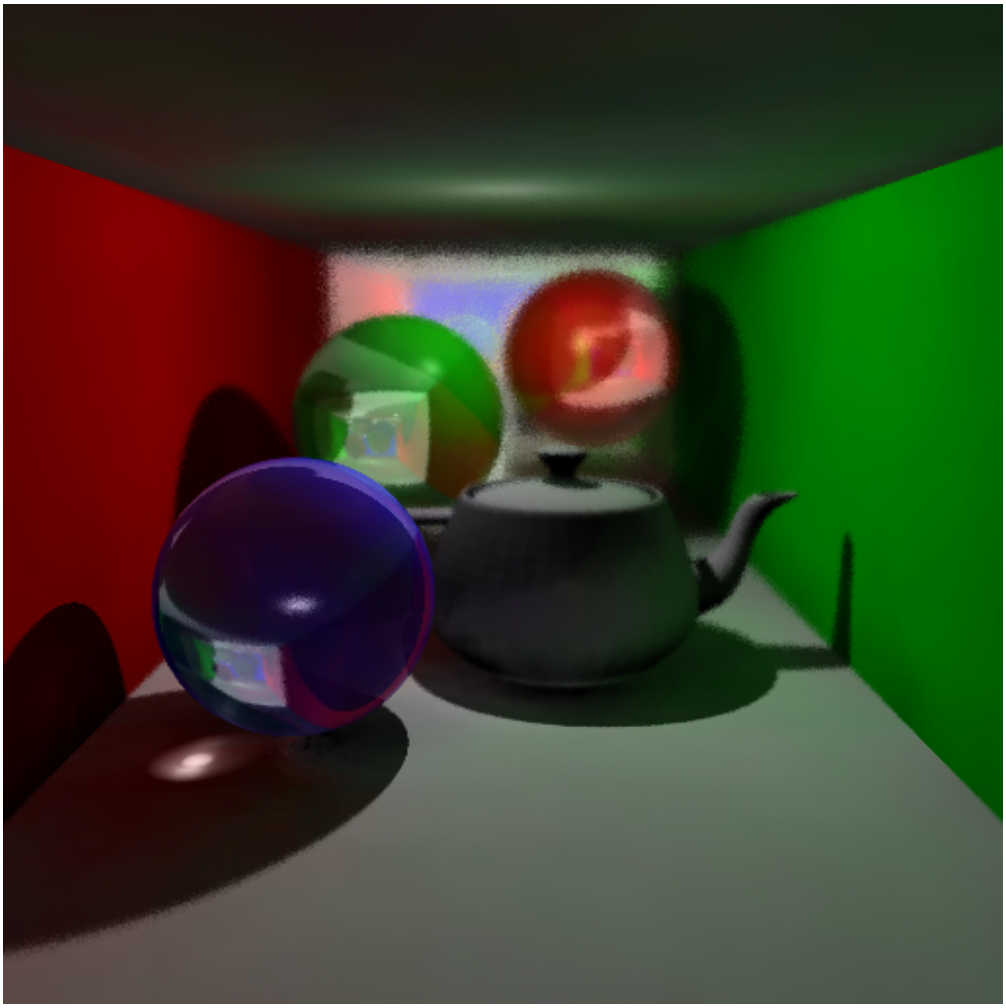
5.1 Gallery



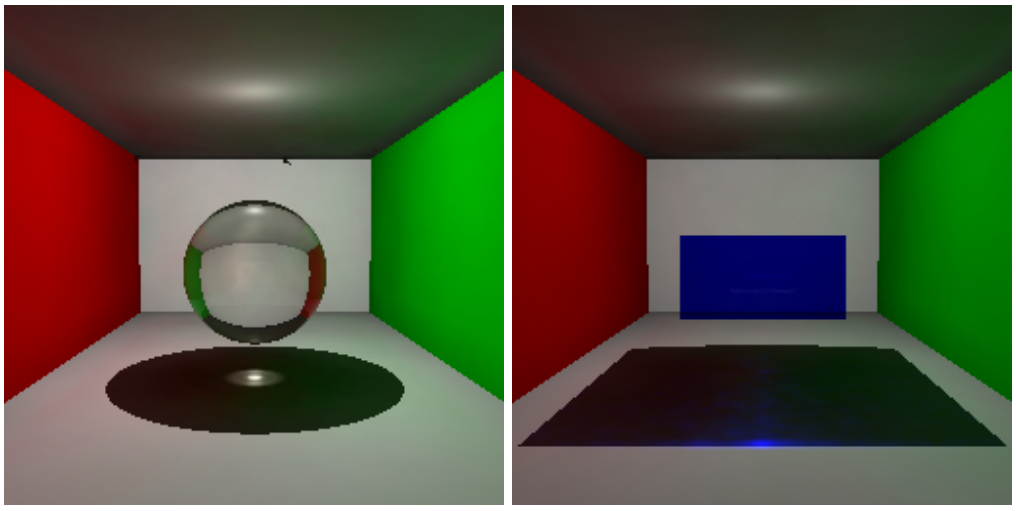
Notice here:

- Sphere is transparent and has properties from Fresnel terms (the wall behind the camera is blue, so blue tint on sphere is from reflections)
- Back wall is reflective
- Caustics from sphere on floor
- Diffuse interreflections from green and red walls on floor and ceiling
- Depth of field effect - notice soft edges of walls in the back of the scene

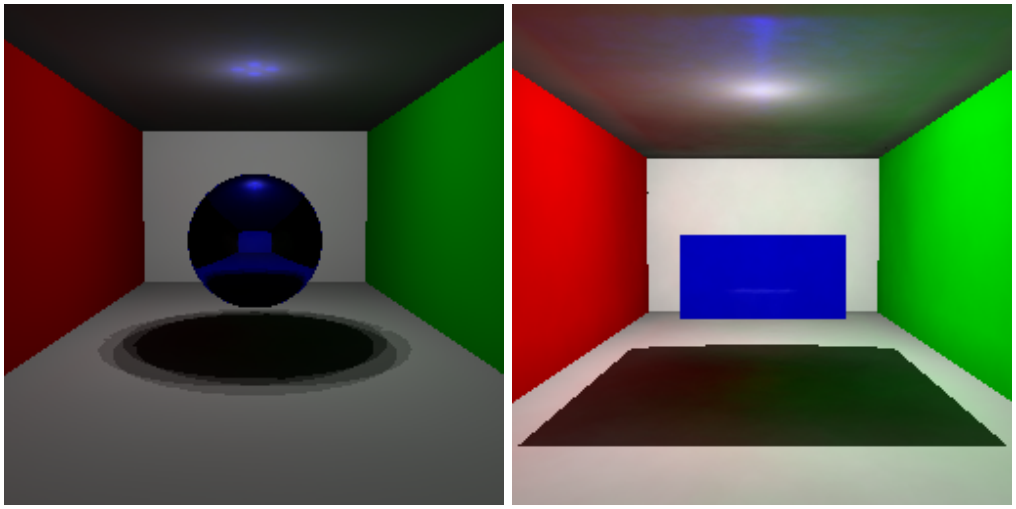
This image shows that all objectives mentioned in section 1 were met.



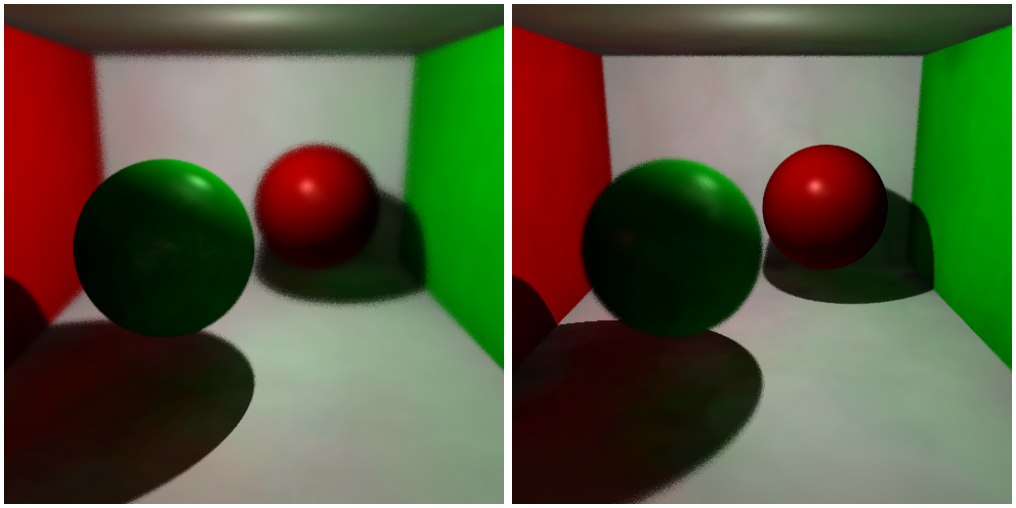
"Cornell Corridor" - first draft of final image. Needs more depth of field samples and/or smaller lens size to have less noise around edges.



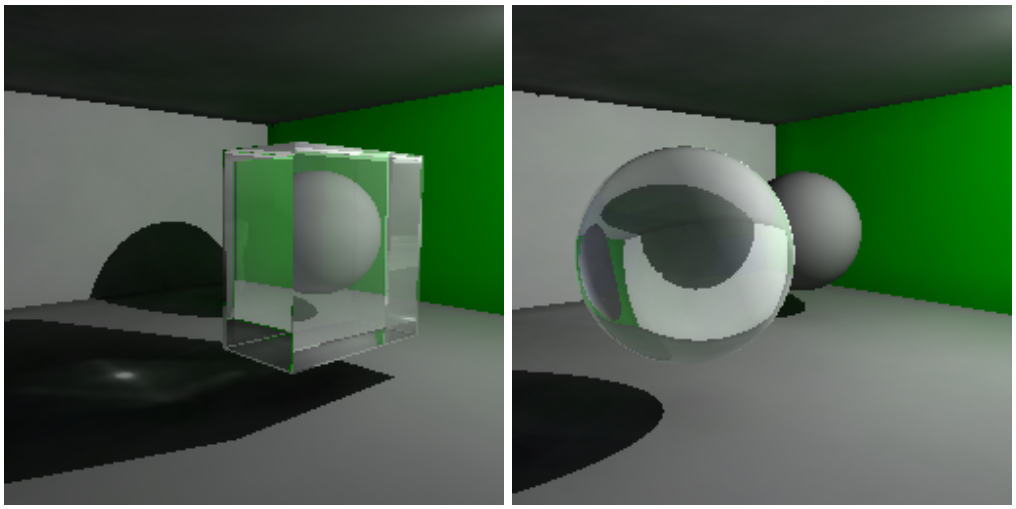
Transparent objects refracting light and creating caustics on the floor. (Left: 70000 caustic photons used, Right: 120000)



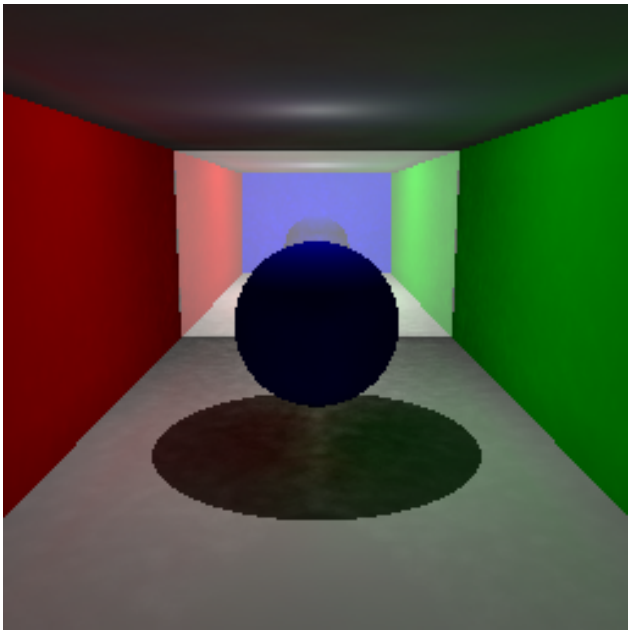
Reflective objects casting caustics on the ceiling. Notice that the light is white, but there is blue light reflected onto the ceiling from the objects. For the left image, 4 point lights were used in the scene.



Depth of field effect, with focal plane set approximately at the nearest sphere / farthest sphere respectively.



Transparent objects with index of refraction 1.52, showing effect from Fresnel Equations (Left: the wall behind the camera is blue - notice the slight blue tint on the left side of the cuboid; Right: notice green tint on right side of sphere) and total internal reflection (Left: see edges of cuboid). Notice also the caustics in the left image.



Reflective surface at the back of the scene. Notice also some caustics on the floor.

5.2 Usage

To change the scene, edit the main.cpp file:

- Change sceneSize to match amount of objects
- Define objects in createScene and assign them to elements in the scene array
- Change lightCount to match amount of lights
- Add lights in main() function (currently only point lights work with photon mapping)
- Change photon totals appropriately
- Adjust the value passed to setCausticBrightness() such that caustics are visible (and scene does not get dark in comparison)

References

[Jensen, 1996] Jensen, H. W. (1996). Global illumination using photon maps. In *Eurographics workshop on Rendering techniques*, pages 21–30. Springer.

[Jensen and Christensen, 2000] Jensen, H. W. and Christensen, N. J. (2000). A practical guide to global illumination using photon maps. *SIGGRAPH 2000 Course Notes CD-ROM*.

[Kajiya, 1986] Kajiya, J. T. (1986). The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 143–150.