CM30075 Advanced Computer Graphics: Report

Candidate 21532

December 2022

Contents

1 Overview **2**
1.1 Ray tracing . . . 2
1.2 Language and Libraries . . . 2
1.3 Structure . . . 2

2 Basic Ray tracer **3**
2.1 Camera . . . 3
2.1.1 Multisampling . . . 3
2.1.2 Multithreading . . . 3
2.2 Basic Objects . . . 3
2.3 Lighting . . . 4
2.3.1 Lights . . . 4
2.3.2 Occlusion . . . 4
2.4 Materials . . . 4

3 Feature-Rich Rendering **4**
3.1 Reflection and Refraction . . . 4
3.1.1 Fresnel Term . . . 5
3.2 Constructive Solid Geometry . . . 5
3.3 Quadratic Surfaces . . . 5
3.4 Depth of Field . . . 5
3.4.1 The Thin Lens model . . . 6

4 Photon Mapping **6**
4.1 Photon Generation . . . 6
4.1.1 Events . . . 6
4.1.2 Dielectric shadows . . . 6
4.2 Photon Map . . . 6
4.2.1 Caustics . . . 7
4.3 Rendering . . . 7

A Images **9**

1 Overview

1.1 Ray tracing

Ray tracing is a method used in computer graphics to simulate light moving through a three-dimensional scene. This report accompanies a piece of software that uses ray tracing to produce an image using the famous Utah teapot model.

1.2 Language and Libraries

Rust The software is written in Rust. Rust was chosen for this project because of its speed and support for multi-threading. It is also fairly similar to the more commonly-used C++, making the translation of the provided starter code fairly simple.

glam Glam is a library that provides types and methods for vectors, a crucial foundations of graphics software. All types are also safe to use in parallel, perfect for ray tracing, and there is also a type for 3D transformations.

png The software writes the pixel data produced into a .png image file. The png library allows this, and is easier than writing to a .ppm file and then converting it manually, despite the simplicity of the .ppm format. The library also handles the complexity of properly adjusting the colours if the image produced is too bright or dark.

acap "As close as possible", abbreviated to acap, is a library that provides a number of different data structures which support search by proximity, including a KD tree. The library also includes methods for balancing and searching automatically.

1.3 Structure

The program is separated into multiple files, grouped by the types and traits that they contain. For types that implement a trait, their files are arranged into a folder named after the trait and the trait definition is in a file directly in the `src` folder. For example, the trait `Object` is defined in `src/object.rs` and the type `Sphere` that implements `Object` is defined in `src/object/sphere.rs`. Some files, like `src/photonmap.rs`, define multiple related types.

To run the program, you must have Rust installed: go to `https://rustup.rs/` to do so. Then the command `cargo run --package raytracer --release` may be used. The program will produce .png files as output in the root folder of the project.

The file `src/main.rs` contains the main function, which initialises the frame buffer, scene and camera before rendering the scene and photon map. Finally, it calls the buffer to write its data to a file. This file also contains the function that creates the scene by specifying objects and their materials, as well as lights. See a summary of the contents of each file in `src/` in Table 1. Files inside folders in `src/` are excluded for readability.

Table 1: Project files in `src` folder

File	Contents
main.rs	Main control of program flow and scene building
lib.rs	Project modules and Vertex type alias
colour.rs	Colour type and associated functions and operators
framebuffer.rs	FrameBuffer and Pixel types, and file writing functions
fullcamera.rs	FullCamera type and functions, including scene rendering
hit.rs	Hit struct which stores information about ray-object intersections
light.rs	Light trait definition, links to 'light/'
linedrawer.rs	**[unused]** LineDrawer type and functions, used during development to debug Polymesh object
material.rs	Material trait definition, links to 'material/'
object.rs	Object trait definition, links to 'object/'
photonmap.rs	PhotonMap, Photon and PhotonHit types, functions and enums, including PhotonMap creation
ray.rs	Ray type and functions and Reflectable trait
scene.rs	Scene type and functions, including querying objects for their interactions with Rays

2 Basic Ray tracer

2.1 Camera

In order to produce an image of a scene, a camera is used. The camera may be moved and rotated, its aperture adjusted, the number of samples per pixel altered, and an image size given. The camera position serves as the origin point for the rays that are fired through the image plane and into the scene. Each ray returns a colour which contributes to the final colour of the corresponding pixel it passes through.

2.1.1 Multisampling

To create a more accurate image, multiple rays (samples) are fired through each pixel, and they all contribute equally to the final colour, as described in [1]. The rays are fired with a small element of randomness to prevent them all taking the same path. Multisampling also allows the number of rays in the scene to be controlled. Although there are a greater number of rays per pixel, rays can be reflected or refracted (Section 3.1) stochastically (also called "Russian roulette" in the context of photons) rather than "branching", which generates multiple new rays from an intersection. Generating multiple new rays can result in a much greater quantity of them in the scene, slowing down the runtime considerably.

In addition, this technique allows for depth of field in the image (Section 3.4).

2.1.2 Multithreading

The camera runs a function 'render()' to produce the data for each pixel. The image is divided into 8 sections vertically so that 8 threads can run potentially at the same time on 8 cores, speeding up the rendering time massively. Rust was built to make parallel computing easy, and ray tracing lends itself well to parallel execution. Memory safety, mutexes etc. are all taken care of by Rust, and provide a huge optimisation. When a thread has computed a colour for the pixel it is working on, it send the pixel position, colour and depth to a receiver via a channel, and the receiver picks each pixel off the channel and writes it to the framebuffer. After all the sending threads finish, the receiver thread finishes too, and the framebuffer contains the complete image.

2.2 Basic Objects

The software supports a few different types of object. An object, no matter its type, must be able to determine if a given ray will intersect with it, and if so, where the intersection will be, what event happens at the surface (ray is reflected, absorbed or transmitted), and so on.

Spheres The most simple is the sphere, since the test for an intersection is easy. The parametric equation of a sphere is used, along with the ray, to determine whether there is an intersection and its position. Spheres are commonly used to showcase different materials in graphics, as they have uniform shape with varying surface normals.

Plane An infinite plane can be expressed as a surface normal and a point in the plane. Using the implicit equation, the intersection test is very basic geometry. The plane is useful during debugging to show the shadow of other objects, which places them in relation to one another.

Polymesh A polymesh is a set of primitives (triangles) that together form a model, with famous examples being the Stanford bunny, Utah teapot and Stanford dragon. The software is capable of interpreting polymeshes from files in the "kcply" format (as opposed to the standard .ply format) that are zero or one-indexed. Each triangle is defined as a trio of vertices in the file. The intersection testing for a polymesh is intensive and complex, as it includes smoothing by using vertex normals. In order to achieve this, normals for each triangle and vertex are computed during the creation of the object, requiring several passes through the triangles.

Intersections with triangles are computed by using Barycentric coordinates and the Möller-Trumbore intersection algorithm [4]. This allows the vertex normals to be combined at a location where the ray hits according to the proximity of the intersection with each vertex, producing a smooth looking object in the output. The program uses the Utah teapot data to showcase this. This intersection test is time consuming, as every single triangle must be checked for an intersection with every ray, but produces a nicer image. This can be seen in Image 1.

Triangles A single triangle can be used to form a plane with finite size using the same intersection algorithm as the polymesh, which is useful for when it's not worth the time to write a whole file in kcply or ply format. The triangle objects are used in the program to form the walls of a Cornell box.

2.3 Lighting

2.3.1 Lights

Lights need to be used in order to realistically colour objects and their shadows in 3D space. A light emits light in one or many directions of some given colour. The light colour will alter the the colour that appears on surface in the image.

The most basic type of light is the directional light, which simulates a light source that's very far away (like the sun). The direction from any point on any surface is always the inverse of the light's direction. The problem with using a directional light is the lack of control over the light's position. A point light is a point in space that emits light equally in all directions. This is excellent for scenes such as the Cornell box, where the walls and ceiling of the box would occlude a directional light from above.

The traditional Cornell box uses an area light, which is yet more realistic than the point light. However, using an area light incurs significant computational cost, and is out of the scope of this project.

2.3.2 Occlusion

In order for shadows to be more realistic, there needs to be some way to make sure that when an object comes between a surface and the light source, the object casts a shadow. This is called occlusion, and is controlled by the `Scene` using the `shadow_trace()` function.

2.4 Materials

The appearance of an object is affected by the light that hits it interacting with its material. The material controls how metallic or glossy a surface is. One material that tends to be used for debugging that does not interact with incoming light is the normally shaded material, where the colour output depends on the surface normal. It is useful especially when developing the polymesh object, as the surface normals are easy to get wrong. During development, I often ended up with inverse normals or normals of non-unit length, and the normally shaded material makes it very easy to debug that issue. This can be seen in Image 2.

Other materials include metal, glass (dielectric), matte and combination materials. Incoming light gets reflected, transmitted or absorbed, according to the material, and for each ray fired at a material, a colour is returned. Since this process is recursive for reflective objects like metal (as the pixel colour needs to have contribution from the objects in the reflection, too), there is a maximum recursion depth. As mentioned above, in situations where a ray may reflect or refract, the outcome is determined based on probability. Reflection and refraction are not features of the basic ray tracer and are discussed in Section 3.1.

Diffuse (matte) surfaces scatter incoming rays in a random direction, resulting in a uniform surface that is not glossy or specular. This is often called Lambertian, and is one part of the Phong lighting model [5]. The Phong material was developed to approximate light more accurately than previously available by combining three terms: ambient, diffuse and specular. The result is a plastic-like material that shows the form of the object. The ambient term is the darkest colour, and applies to the entire object. The diffuse term provides the mid-tone, depending on the location of the light sources. The specular term depends on the position of the camera (viewer) in relation to the surface, and is the brightest term. The Phong lighting model may look unrealistic by modern standards, but was groundbreaking in its time.

3 Feature-Rich Rendering

3.1 Reflection and Refraction

Reflection and refraction occur when a ray hits a material that is metallic or dielectric (glass-like). Reflection occurs when a ray hits a reflective surface, at which point a new ray is created from the hit point in the reflected direction about the surface normal. If that ray hits another reflective surface it may spawn yet another new ray until the

recursion depth is reached (to prevent infinite ray bouncing). If the ray hit a diffuse surface, the surface colour contributes to the initial hit (as if an object is in a mirror). For a metallic material, all rays are reflected, but the direction may be affected by the `roughness` field. The rougher the material, the more the direction of the reflected ray is altered by randomness. This is seen in the `interact()` function on `Metallic`:

```
let r = (hit.incident.direction.normalize().reflect(hit.normal)
        + self.roughness * random_in_unit_sphere())
    .normalize();
```

Refraction occurs according to Snell's Law : the angle of a light ray changes when crossing the boundary between media with different refractive indices. Snell's Law is given by: $\frac{\sin\theta_1}{\sin\theta_2} = \frac{n_1}{n_2}$ where θ is the angle of incidence and n is the refractive index of the material.

3.1.1 Fresnel Term

When an observer views a dielectric object directly (direction is close to opposite the surface normal), they can see straight through it. For example if you stand in a swimming pool and look down, you can see through the water. However, if you look out across a lake or ocean, you cannot see through it: this is due to the angle of observation being close to perpendicular to the surface normal. In this example, a large body of water acts like a mirror when viewed in this way. Fresnel determined exactly how the ability of a surface to reflect light varies with incident angle.

This phenomenon can be simulated by introducing the Fresnel equations into the `Dielectric` material as described in [7], but a good approximation may be used instead: Schlick's approximation [6] for reflectance. The following function returns the reflectance probability of the material given the cosine of the incident ray and refractive index:

```
fn reflectance(cos: f32, index: f32) -> f32 {
    let r0 = ((1. - index) / (1. + index)).powi(2);
    r0 + (1. - r0) * (1. - cos).powi(5)
}
```

This effect can be observed in Image 3. Notice how in the right image, around the sphere edges, there is a faint reflection of the environment visible.

3.2 Constructive Solid Geometry

CSG is a technique used very often in modelling to create complex shapes from basic 3D primitives. A CSG is a binary tree of objects and their relationships. For example, two intersecting spheres may have the operation `Op::Union`, resulting in a single object that combines the areas bound by both. The relationship between objects can be `Op::Union`, `Op::Intersection` or `Op::Difference`. These three cases are shown in Image 4.

3.3 Quadratic Surfaces

Quadratic surfaces are surfaces defined by the following: $Ax^2+By^2+Cz^2+Dxy+Exz+Fyz+Gx+Hy+Iz+J=0$ and are most commonly an ellipsoid, cone, paraboloid or hyperboloid. Alternatively, they can be expressed as:

$$\begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} a & b & c & d \\ b & e & f & g \\ c & f & h & i \\ d & g & i & j \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = 0$$

The `Quadratic` object stores coefficients a–j as the array `coeffs` instead of as a $4 * 4$ matrix to prevent redundancy. Admittedly, this makes the intersection test and transformation very verbose. With more time, there is definitely room to make this object implementation more streamlined. An example of a common quadratic surface case is seen in Image 5: an ellipsoid.

3.4 Depth of Field

Depth of field refers to the fact that some objects appear sharp and others appear blurry, according to the distance of the objects from the camera's focal plane. This cannot be observed when the camera acts as a pinhole camera with a infinitely small aperture. Depth of field makes an image appear more realistic, simulating a non-pinhole camera, and the strength of the blur produced is controlled by the camera's aperture. A pinhole camera may still be simulated by using an aperture of 0.0.

3.4.1 The Thin Lens model

The "Thin Lens" model in the field of optics makes some generalisations about the physics of a real lens by assuming the lens' width is negligible compared to the distance from the lens to the focal plane. The focal plane is the same as the image plane in this implementation, as it locates the pixels of the final image, which should be in focus, relative to the 3D scene. The lens is implemented as a circular disc around the camera's position within which the rays are fired from. The greater the aperture, the more spread out the rays travelling towards a given pixel are, and the less "accurate" the final colour will be if the colour comes from an object far away from the focal plane.

The offset of rays by some amount within the lens disc is achieved by generating a random vector within a unit disc and multiplying it by the camera's aperture. The direction of the ray also needs to be adjusted, otherwise it will not point exactly to the correct pixel, and objects that should be in focus would appear blurry. The correct effect can be seen in Image 6, where the focal plane intersects with objects roughly in the middle of the box.

4 Photon Mapping

Photon mapping is a technique used to more accurate simulate real light by tracing rays (photons) from light sources into the scene and then using the resulting information about their paths in the rendering stage, as described in [2]. By using photon mapping, some important visual effects can be shown in renders, like caustics through a glass object or colour bleeding from matte surfaces.

This program uses multiple photon-related types for this, including `PhotonMap`, `PhotonHit` and `Photon`. There are also relevant methods added to `FrameBuffer` and `Material`. Photon mapping is done in two stages and is thus called a two pass algorithm: first the `PhotonMap` is generated,a nd then it is used during the rendering stage in conjunction with the more basic ray tracing methods.

4.1 Photon Generation

The first pass is the generation and firing of `Photons`. Photons are fired from each light source into the scene which, in the case of the `Point` light, means in all directions. Photons that hit an object are recorded as `Type::Direct` photons in a `PhotonHit`. Depending on the material of the object, a new photon might be generated that is `Type::Indirect` - this occurs when a photon hits an object and is `Interaction::Reflected` or `Interaction::Transmitted`. For each direct hit, a `Type::Shadow` photon is also fired along the same trajectory as the original photon, in order to record the location where that direct hit casts a shadow. This is used for optimisation during rendering.

4.1.1 Events

The type of event that occurs when a photon hits a material is recorded with the Enum `Interaction`, which is returned when the `interact()` function is called on a material. It is specifically useful for matching against when each type of event is treated differently. For a material like `Dielectric`, the interaction could be transmission or reflection, determined stochastically.

4.1.2 Dielectric shadows

A side effect of introducing the different types of interactions with materials, the `shadow_trace()` function was augmented to increase the accuracy of glass shadows, as seen in Image 7. If a ray is transmitted, no shadow is produced: this means that the glass sphere produces a shadow that shows off the Fresnel behaviour with a darker ring where rays are reflected around the edges.

4.2 Photon Map

Photon hits are stored in a photon map in order to be accessible during the rendering phase. The preferred data structure for this is the k-d tree. This is used because "In order for the photon-mapping algorithm to be practical, the data structure has to be fast when it comes to locating nearest neighbors in a three-dimensional point set." [3]. A great library for this is the `acap` library, containing the `KdTree` type. The library has support for a "k-closest neighbours" search, perfect for photon mapping. A visualisation of a complete photon map is seen in Image 8, which uses the same scene as Image 7. This visualisation shows a sort of irradiance estimate for the scene by finding the closest photons in a radius around the point in question and taking an average of their intensity. The more photons you fire, the more accurate the photon map is, but the longer it takes to construct.

4.2.1 Caustics

Caustics are generated separately from the normal photon map, as a very large number of photons is required to create a convincing effect. A caustic photon map is treated slightly differently in the code, for example by using the `build_caustics()` function instead of the `build()` function. Caustics occur at refractions or specular reflections, but this program only creates refracted caustics.

The caustic map is used in a different way to the standard photon map in that it can be directly visualised instead of meshed into the rendering phase. This code achieves this by creating a normal rendered frame buffer and combining it with a frame buffer containing the visualisation of the caustics directly with the `add_caustics()` function of `FrameBuffer`. This effect is seen for a large glass sphere in Image 9.

4.3 Rendering

The following is not implemented in the submitted software, and the only "rendering" is the naive addition of a caustic photon map, which is very limited in practice.

The photon map(s) are used during the rendering stage to produce a more accurate image. Different types of ray path are handled differently, and are rendered either accurately or inaccurately depending on how much they contribute to the image. The types of ray path are distinguished according to the types of interactions photons have on their path from the light: photons that have hit specular surfaces and been reflected are treated as a separate case to a photon that has directly been absorbed from the light.

The photon map provides an estimate for the amount of energy (radiance) entering and leaving each part of the scene. This is computed by integrating over an area of some radius centering on the point in question, according to the hits recorded in the photon map. This is similar to the method used to directly visualise the caustics map described above. All of this serves to complete each component of the rendering equation, which "forms the mathematical basis for all global illumination algorithms" [3], given as:

$$L_o(x, \overrightarrow{\omega}) = L_e(x, \overrightarrow{\omega}) + L_r(x, \overrightarrow{\omega})$$

This equation describes the outgoing radiance at a point (L_o) as the sum of the emitted and reflected radiances (L_e and L_r). The emitted radiance for objects that do not give off light is zero, so we just need to work out the reflected term, which can be expressed as the sum of four integrals that are evaluated separately. The four contributions are: direct illumination from lights, specular reflections, caustics and indirect illumination (photons that have bounced diffusely at least once) [2].

Direct illumination is computed accurately (without the photon map) when the photon map indicates that the area in question is directly illuminated and there are no shadow photons near-by. Inaccurately, it can be taken directly from the radiance estimate by taking photons from the photon map.

Specular reflections are computed without the photon map, as the photon map is not accurate enough to give a nice image.

Caustics are computed using *only* the information in the caustic photon map, as traditional ray tracing struggles to simulate these effects at all (although it is possible).

Indirect illumination is where the information in the photon map is really valuable. For non-accurate evaluation, the radiance estimate from the photon map is fine, but for accurate evaluation this needs to be combined with the traditional ray tracing to produce a convincing effect.

The combination of these approaches allow for caustics and diffuse colour bleeding, and well as the soft illumination that comes from diffuse photon bouncing, that are not typically seen in traditional Monte Carlo style ray tracing. A particularly famous application of caustics is the simulation of liquids, as seen in Image 10 [2].

References

[1] Robert L. Cook, Thomas Porter, and Loren Carpenter. "Distributed Ray Tracing". In: *SIGGRAPH Comput. Graph.* 18.3 (Jan. 1984), pp. 137–145. ISSN: 0097-8930. DOI: 10.1145/964965.808590. URL: https://doi.org/10.1145/964965.808590.

[2] Henrik Wann Jensen. "Global Illumination using Photon Maps". In: *Rendering Techniques '96*. Ed. by Xavier Pueyo and Peter Schröder. Vienna: Springer Vienna, 1996, pp. 21–30. ISBN: 978-3-7091-7484-5.

[3] Henrik Wann Jensen. *Realistic Image Synthesis Using Photon Mapping*. A K Peters, Ltd., 2001. ISBN: 1-56881-147-0.

[4] Tomas Möller and Ben Trumbore. "Fast, Minimum Storage Ray-Triangle Intersection". In: *Journal of Graphics Tools* 2.1 (1997), pp. 21–28. DOI: 10.1080/10867651.1997.10487468. eprint: https://doi.org/10.1080/10867651.1997.10487468. URL: https://doi.org/10.1080/10867651.1997.10487468.

[5] Bui Tuong Phong. "Illumination for Computer Generated Pictures". In: *Commun. ACM* 18.6 (June 1975), pp. 311–317. ISSN: 0001-0782. DOI: 10.1145/360825.360839. URL: https://doi.org/10.1145/360825.360839.

[6] Christophe Schlick. "An Inexpensive BRDF Model for Physically-based Rendering". In: *Computer Graphics Forum* 13.3 (1994), pp. 233–246. DOI: https://doi.org/10.1111/1467-8659.1330233. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/1467-8659.1330233. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/1467-8659.1330233.

[7] Kevin Suffern. *Ray Tracing from the Ground Up*. CRC Press, 2016. Chap. 28.1. ISBN: 9781498774703.

A Images

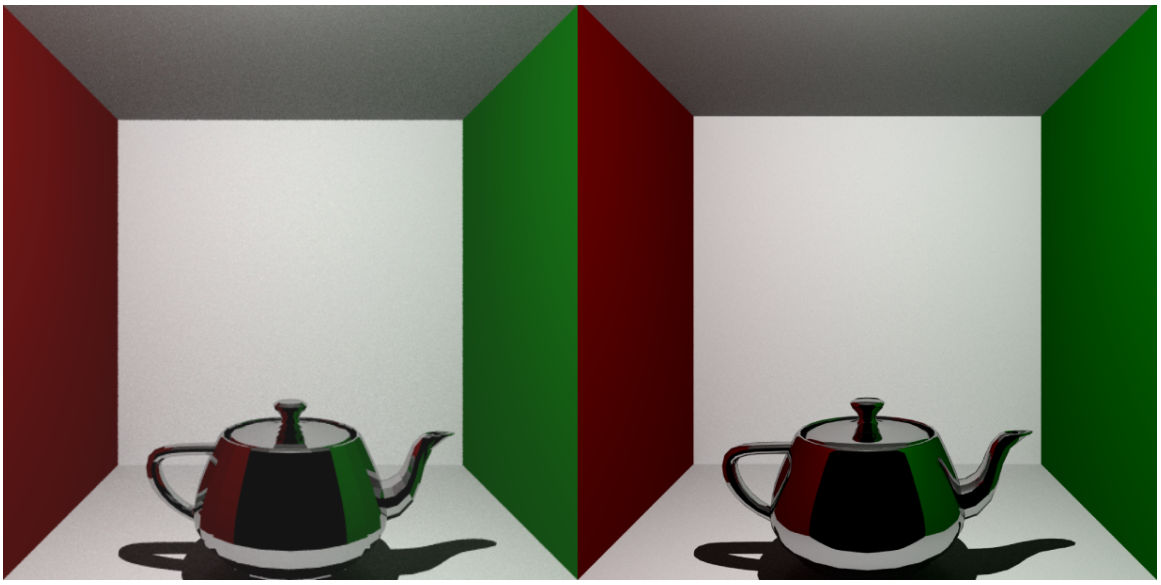

Figure 1: Left: An unsmoothed polymesh. Right: A polymesh smoothed using vertex normals.

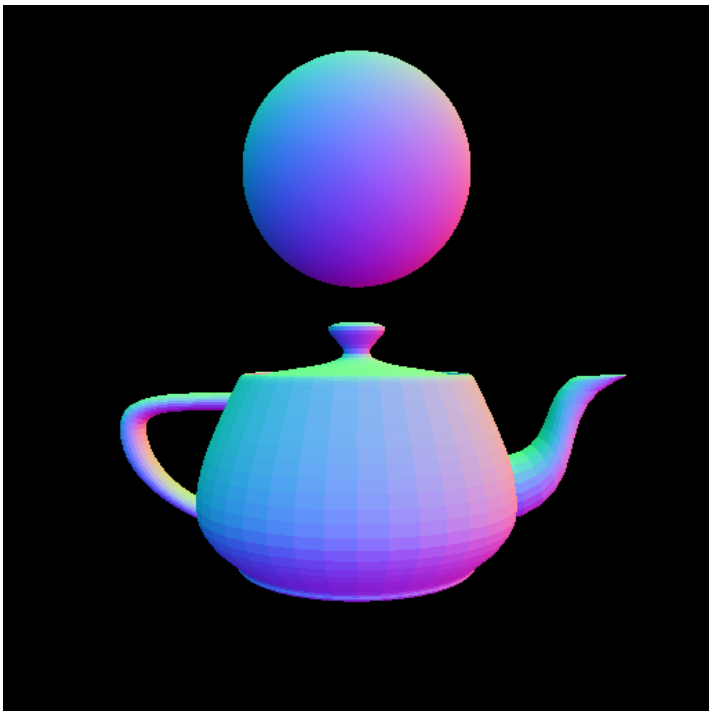

Figure 2: Objects with the normal shading material.

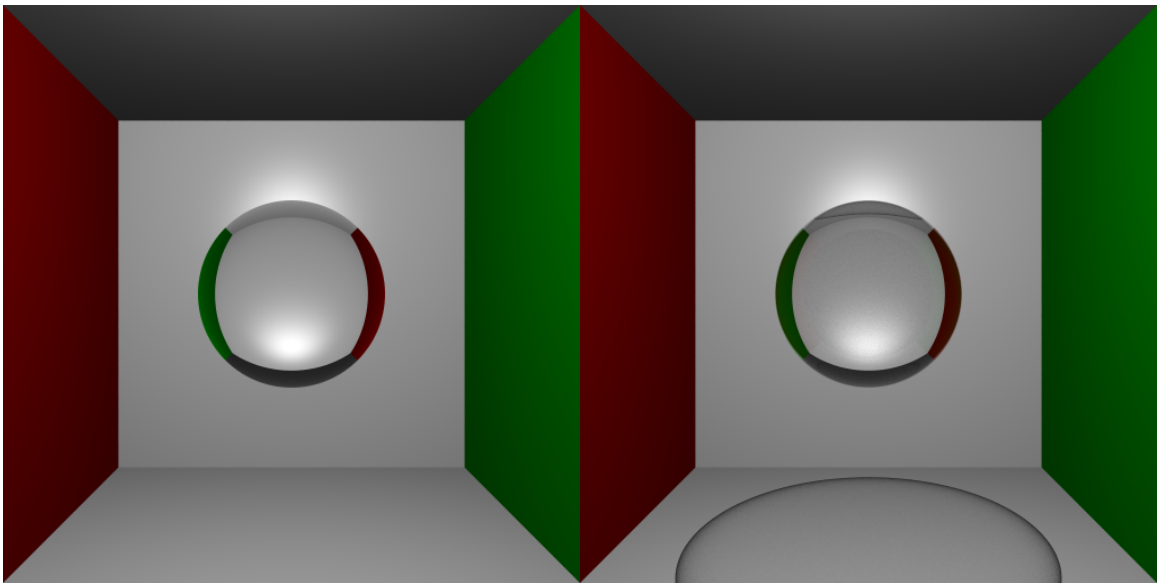

Figure 3: Left: No fresnel term, all rays are transmitted. Right: Fresnel term used, some rays near the edges are reflected and a shadow is seen.

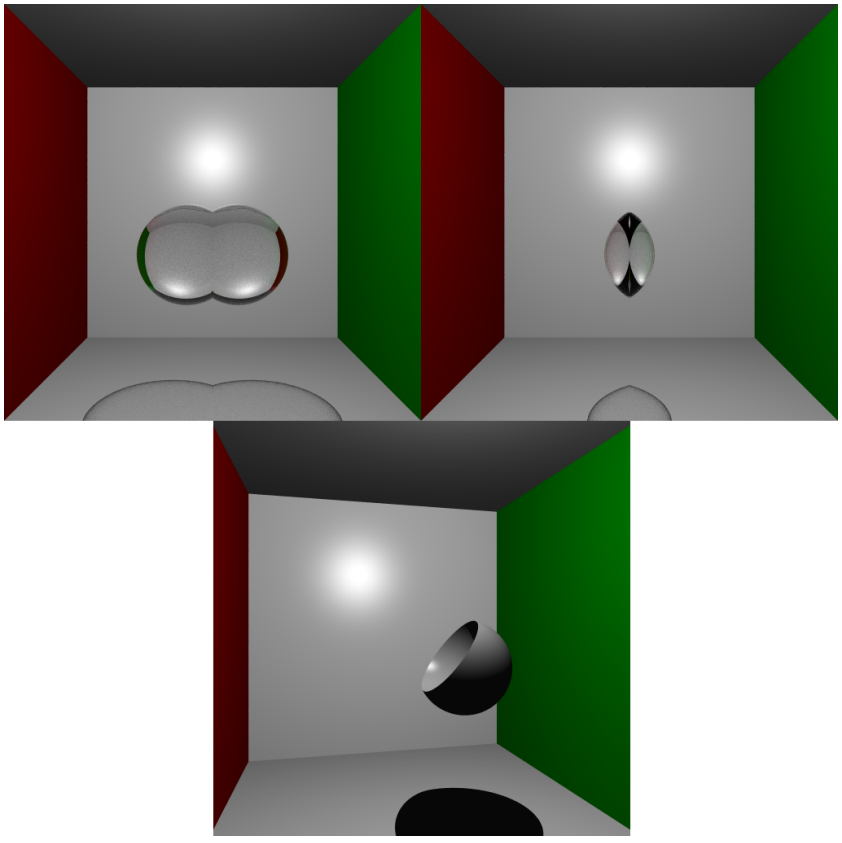

Figure 4: Top Left: Union of glass spheres. Top Right: Intersection of glass spheres. Bottom: Difference of Phong spheres.

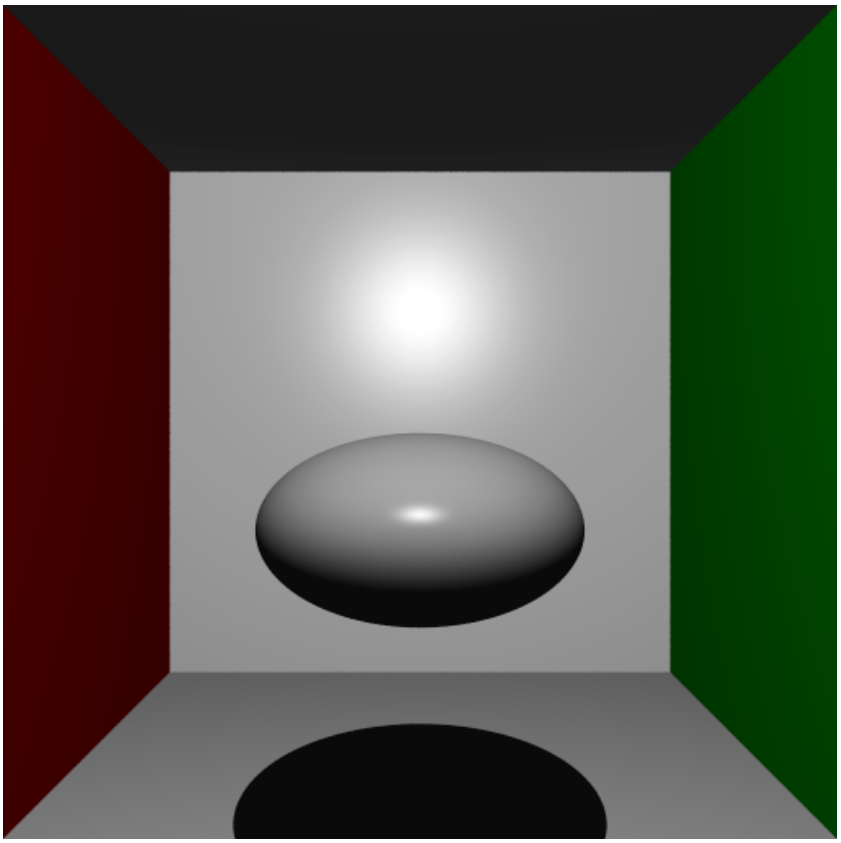

Figure 5: An ellipsoid with a Phong material.

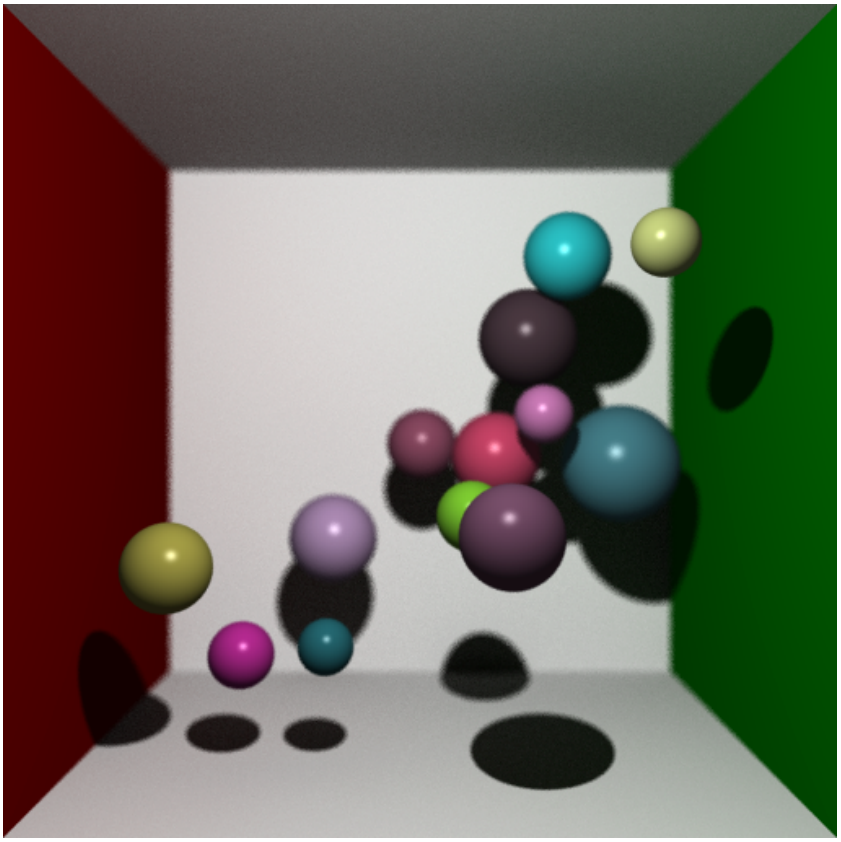

Figure 6: A scene with depth of field: notice how some spheres are blurry and others appear sharp.

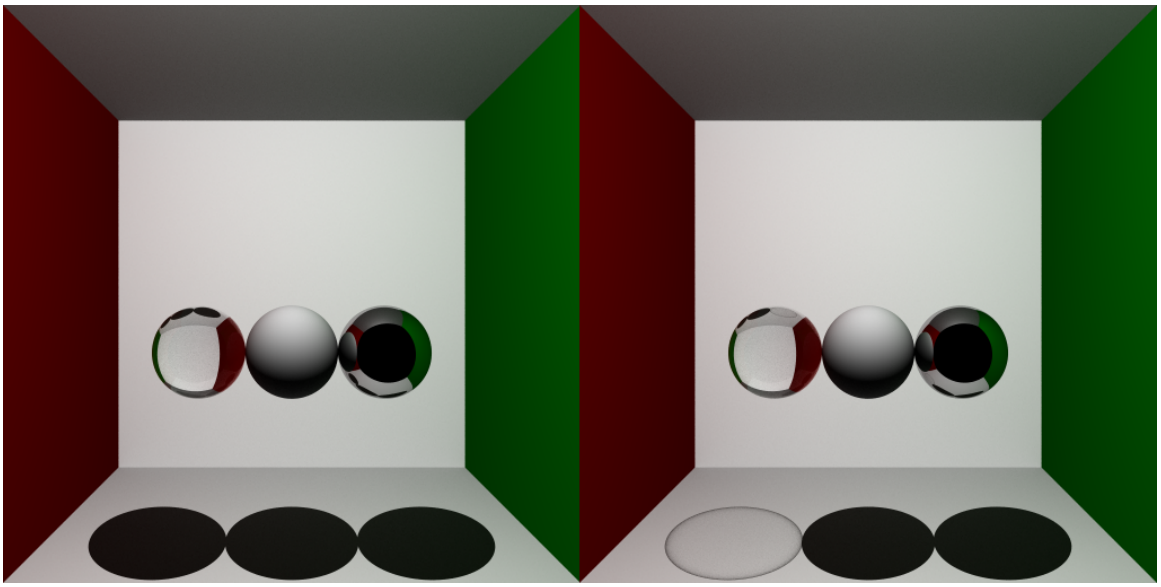

Figure 7: Left: Shadows do not discriminate between materials. Right: Shadows are not created when rays are transmitted by a material.

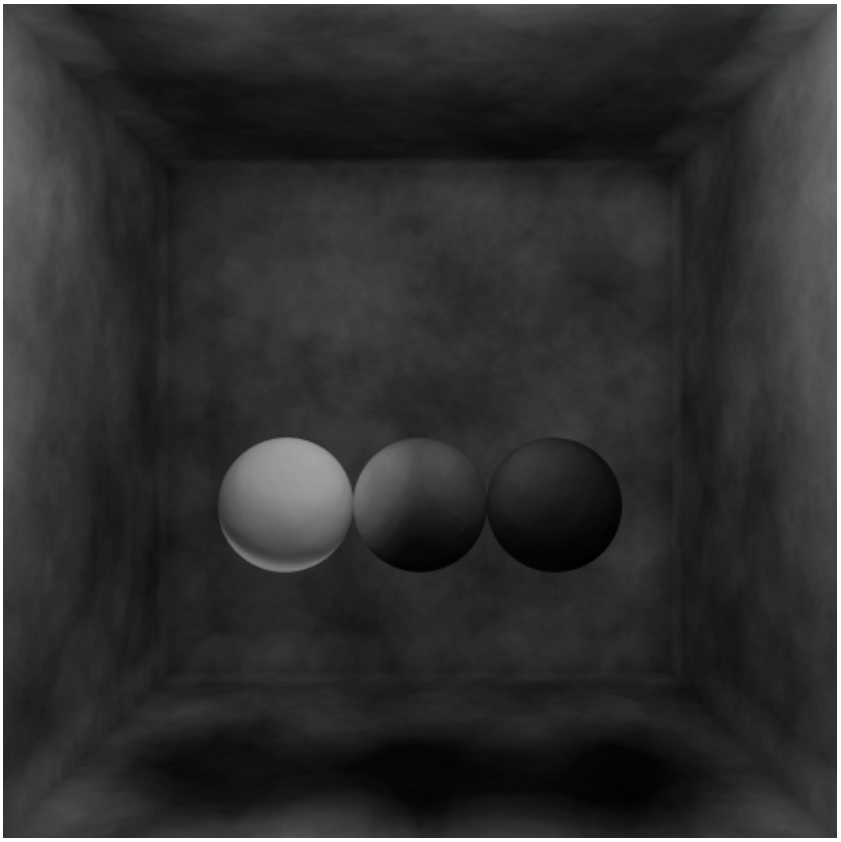

Figure 8: Photon map visualisation.

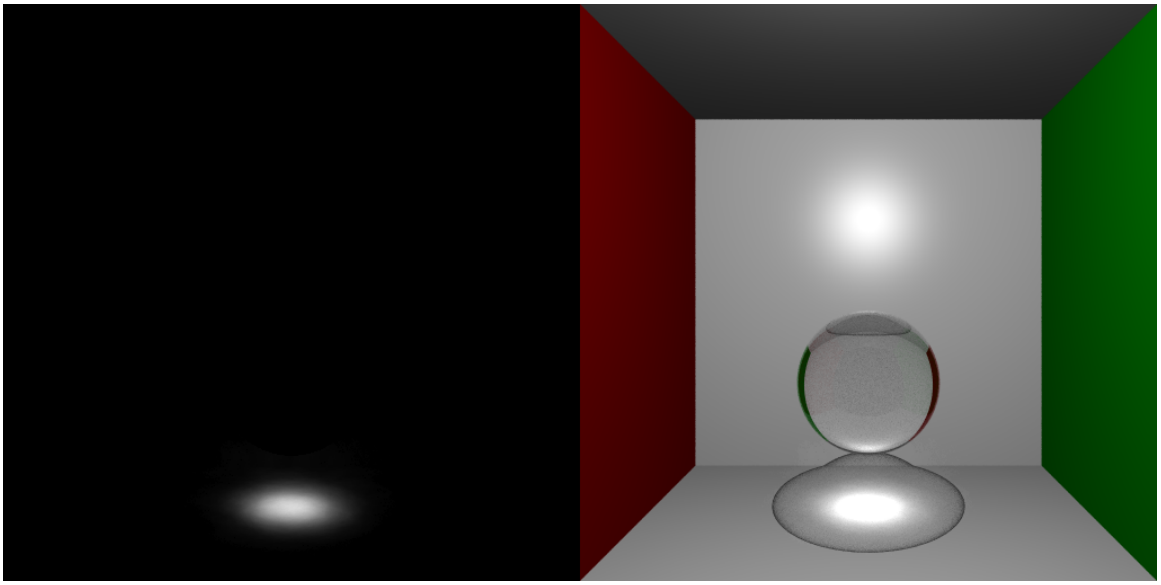

Figure 9: Left: Visualisation of a caustic photon map. Right: Render with caustic visualisation directly added.

Figure 10: A glass of cognac on a fractal surface. The bright parts in the glass' shadow are caustics. Image by Henrik Wann Jensen [2].