Advanced Computer Graphics

CM30075

July 4, 2020

Chapter 1

Introduction

Ray tracing, or more specifically ray casting, is a rendering technique that generates images by modelling light rays fired from a camera into a scene. Intersection tests are carried out for each light ray with each object in the scene, the colour of the pixel is then determined by sampling the colour of the nearest object at the intersection point. Figure 1.1, shows the tracing of rays from a

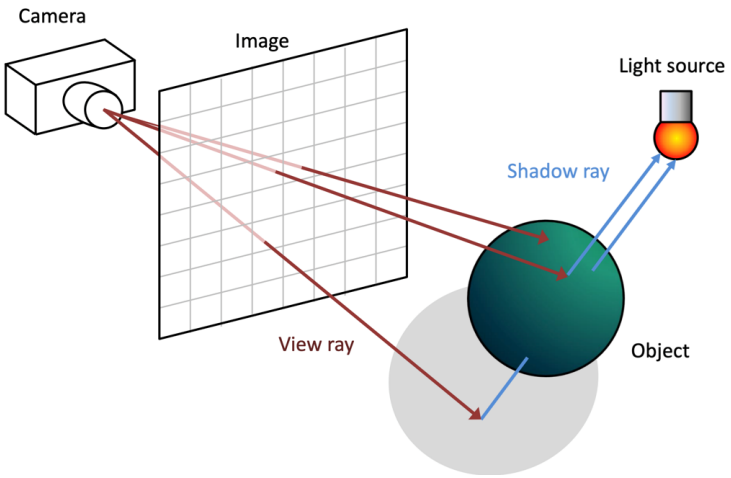

Figure 1.1: Diagrammatic overview of ray tracing (Dubla, 2011)

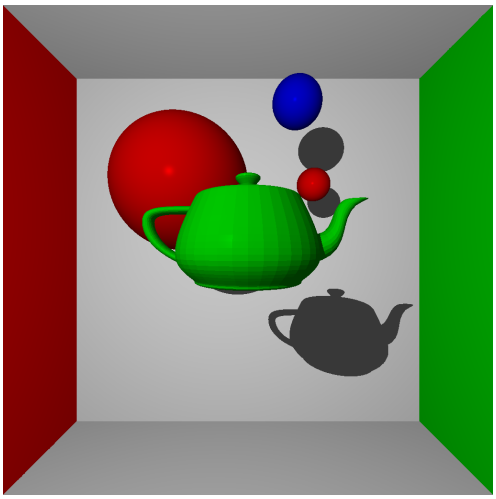

Figure 1.2: Image generated by the ray tracer after completion of lab exercise 4.

camera model, through the pixels on the image plane, to the intersection with an object. Shadows are calculated by firing secondary shadow rays from each point of intersection towards each light source. If the secondary ray intersects an object then it must be occluded, thus in shadow.

An implementation of the Blinn-Phong lighting model is used in combination with shadow ray generation in order to model local lighting (Blinn, 1977). Figure 1.2 displays an image generated from the ray tracer after completing lab exercise 4, before implementing any additional features.

The following chapters detail the extensions made to the ray tracer.

Chapter 2

Global Lighting

Global lighting allows for more convincing images to be generated by modelling more complex lighting scenarios and adding indirect lighting contributions from other other objects in the scene.

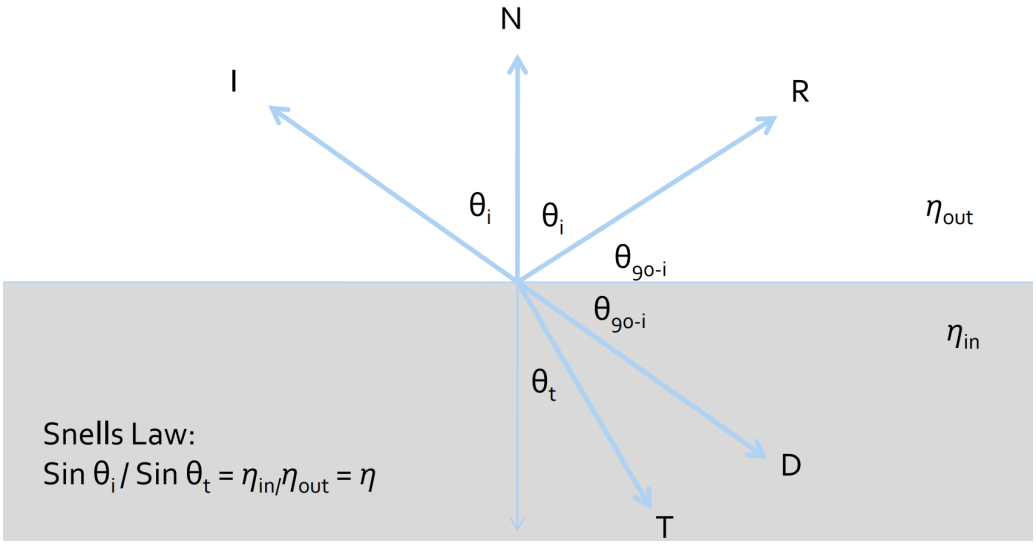

Figure 2.1: Diagram displaying how light rays are reflected or refracted on a given surface with reference to Snell's Law (Cameron, 2019).

2.1 Reflection

To implement reflection, secondary reflection rays are fired from each point of intersection. The direction of the reflected ray is determined by the the angle of the incoming light ray from the camera and the surface normal. The reflection is modelled as a perfect mirror, and so can be computed as follows:

$$R = I - 2(I \cdot N)N$$

Where I is the incident direction and N is the surface normal.

The resultant reflected ray is traced, and the closest point of intersection is

found (if it exists). The colour of the intersected object is sampled and some contribution of that colour is added to the final colour of the pixel, determined by a coefficient kr. This process is then generalised and solved recursively until a given depth is reached, repeatedly firing secondary reflection rays and gathering some contribution of each object's colour at each hit like so:

```
colour += kr * trace(reflection_ray, depth - 1);
```

Figures 2.2, 2.3 and 2.4 showcase the effect of increasing the recursive depth, notice how small amounts of other objects can be seen in the reflection as the depth increases.

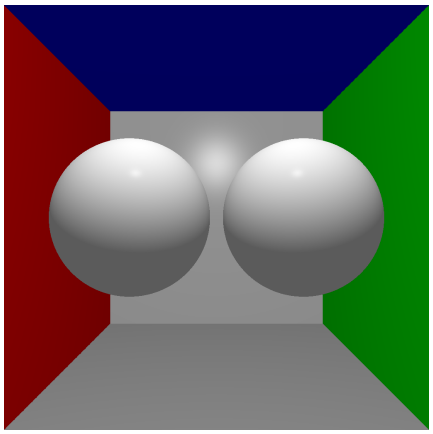

Figure 2.2: Depth 0

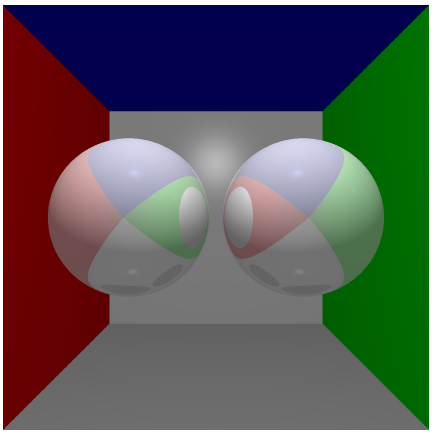

Figure 2.3: Depth 1

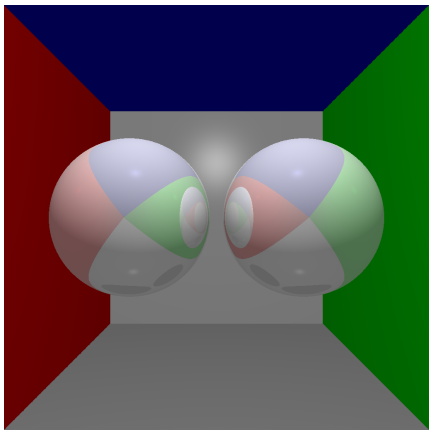

Figure 2.4: Depth 2

Limited floating point precision lead to a "speckled" effect on the surface of the reflective objects, as the hit point is mistakenly considered beneath the surface of the object, leading to self intersection. A similar issue exists with shadow rays, and so in order to mitigate this effect a similar solution was employed. The hit point is shifted by some small epsilon ε in the direction, or opposite direction, of the surface normal depending on if the ray is inside or outside the object.

2.2 Refraction

Refraction can be implemented using a similar method, generating transmission rays from the intersection point and adding some contribution of the resultant intersection with an object to the colour of the pixel. Again, the problem is solved recursively, with the amount of colour to contribute decided by a coefficient kt. The direction of the transmission ray is calculated using Snell's Law, which describes the relationship between the angles of incidence and the angle of refraction, see Figure 2.1.

2.2.1 Fresnel equations

There exists a relationship between values of kr and kt, where: $kt = 1 - kr$. The two coefficients can be calculated accurately using the Fresnel equations, which incorporate the refractive index of the medium to determine exactly how much light should be reflected and refracted (Lvovsky, 2015).

Total Internal Reflection (TIR) is a case where all light for a reflection is reflected back within a medium, no light is refracted. This produces effect such as that shown in Figure 2.6, notice how the insides of the cube appear as perfect mirrors due to TIR. The case of TIR was tested by using the IOR of the medium and Snell's Law to check if the value of $\theta_t >= 1$. If true then TIR should occur, thus $kr = 1$, leaving $kt = 0$. Otherwise, the Fresnel Equations are used to calculate the value of kr, revealing kt.

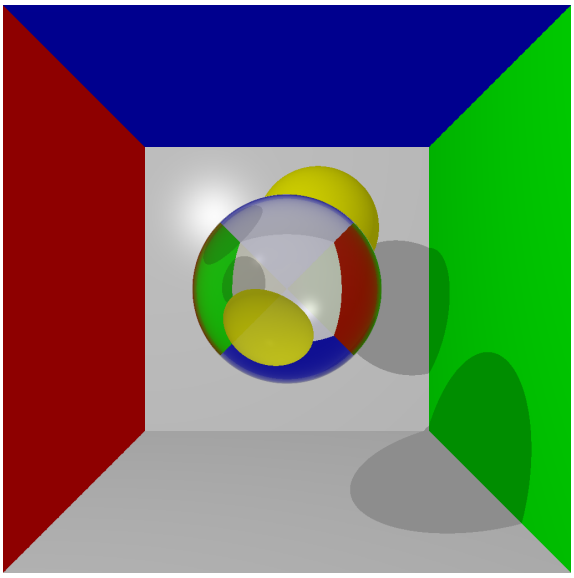

Figure 2.5: Image displaying how light is refracted through a glass ball.

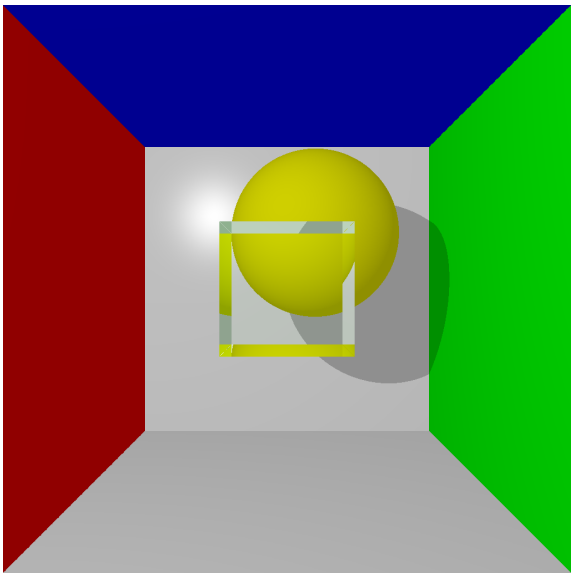

Figure 2.6: Image displaying Total Internal Reflection within the sides of a glass cube.

Chapter 3

Photon Mapping

Photon mapping is technique used to model the behaviour of photons emitted from light sources within a scene (Jensen, 1996). The algorithm devised by (Jensen, 1996) is a two pass algorithm. The first pass distributes the photons within a scene, the second pass renders the scene using conventional ray trace with the inclusion of photons for radiance estimations.

Photons are defined as objects with a light intensity, position, direction, and type. The intensity, position and direction are all determined by the attributes of the light source, and the photon type is determined by the type of tracing occuring. Regular photon tracing uses a type denoted as "default", however other types exist such as "caustic" and "shadow", which are detailed in Sections 3.4 and 3.3.

3.1 Photon tracing

The first pass of the algorithm, photon tracing, was implemented by generating a photon at a given light source and setting its direction to a random direction within the hemisphere of the light. The photon is then traced through the scene. At each intersection with a diffuse surface, the position of the photon is stored. Next, the subsequent direction of the photon is decided and traced recursively. After each bounce the position and intensity are modified. The position is set to the position of the hit, and the intensity is set to the intensity of the object the photon has intersected with. This allows for indirect diffuse light to be captured, as the colour of objects can be "transferred" onto other objects. This tracing process is repeated for n photons.

The next direction of the photon is decided by a russian roulette method, as detailed by (Jensen, 1996). The following code snippet details how this decision is made, where kd, ks and t are the coefficients for diffuse, specular, and transmission respectively:

```
1  float probability = Utils::get_random_number(0, 1);
2  if (probability <= m->kd) { // diffuse reflection
3    ...
```

```
4   } else if (probability <= m->kd + m->ks) { // specular ↩
      reflection
5     ...
6   } else if (probability <= m->kd + m->ks + m->t) { // ↩
      transmit
7     ...
8   } else { // absorbed
9     return;
10  }
```

Diffuse reflection directions are decided by selecting a random point on the hemisphere with respect to the surface normal, whereas specular reflections again behave like perfect mirrors. At each reflection, the photons intensity is boosted by dividing it by its coefficient, this boosts the intensity of indirect light so that it can be seen more clearly. The transmission is handled the same as a refracted light ray, the details of which are covered in Section 3.3.

The stored photons are then used to build a KD Tree using ALGLIB, a popular library, to allow for fast and efficient access during the rendering step. Figure 3.1 displays a direct rendering of the photon map, where the colour of each photon can be seen as small dots within the scene. Notice the indirect colour bleed that occurs on the ceiling due to the result of photons bouncing from the coloured walls of the cornell box.

3.2 Ray tracing

The next step is to render the scene, where conventional ray tracing is used to determine the visibility. In accordance with the rendering equation, the emissive light, direct, indirect, and caustics are all combined to represent the radiance at a given point.

Emissive light is calculate accurately, taking the exact emissive value from the object. Currently, only light sources have an emissive value.

Direct and indirect light are both computed approximately by gathering nearby photons at each intersection point. A radiance estimate can be made by calculating how much light should exist at that position based on the BRDF, the intensity of the photons, and their incoming directions. The calculations involved were implemented in accordance with recommendations by (Jensen and Christensen, 2000).

Nearby photons are gathered by querying the KD tree using a K Nearest Neighbours method. The intensity of each photon is scaled by the output of the BRDF, where the incoming light is represented by the incoming direction of that photon. The BRDF used was a simple implementation of the phong lighting model, as detailed earlier in the report. The scaled intensity value is then run through a gaussian filter in order to weight the intensity of the photon based on its distance from the hit point.

The addition of the gaussian filter results in an overall smoother and cleaner image, as nearby photons are considered more important. The calculated values for all photons are then summed, and scaled by the area of the circle they were sampled from to produce an overall estimate for the radiance at that point.

The following code snippet details how this was implemented:

```
...
vector<Photon *> local_photons = gather_photons(...);

// find max distance of all photons
float max_dist = ...

for (Photon *p: local_photons) {
    float dist = (p->ray.position - hit.position).magnitude();

    Vector base_colour = m->compute_base_colour(hit);

    // gaussian filtering as using coefficients recommended by the paper.
    float gaussian = ...;

    // calculate photon contribution based on brdf and photon intensity
    colour += p->colour * m->compute_light_colour(...) * gaussian;
}
// divide through by max volume of disc sampled within
colour = colour / (M_PI * max_dist * max_dist);
```

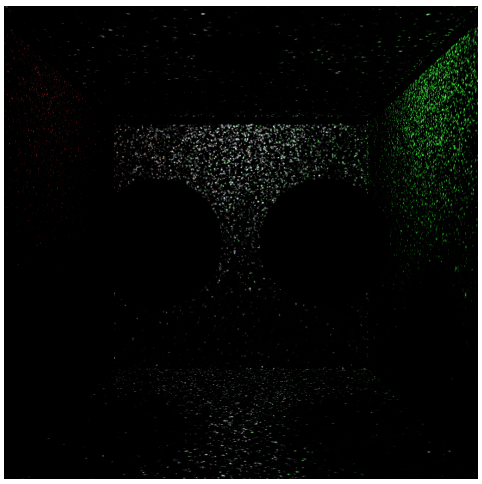

Figure 3.1: Visualisation of the photon map stored within the KD Tree.

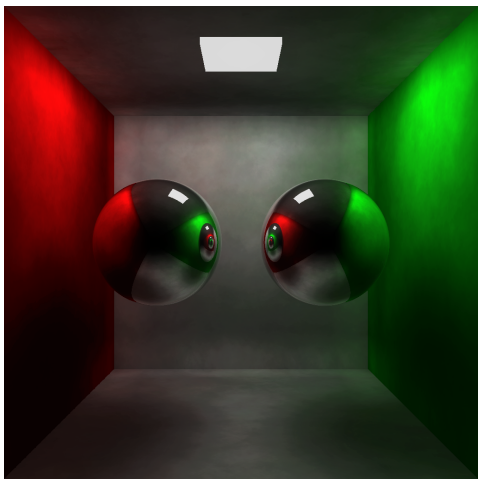

Figure 3.2: Rendering of the scene using the photon map to estimate radiance.

The result of this radiance estimation can then be used as a base colour for the aforementioned local and global lighting calculations, allowing for images such as Figure 3.2 to be generated. Notice how the rear wall is illuminated by indirect diffuse light from the green and red walls. The soft shadows are a result of randomised start position on the surface of the area light.

Something worth mentioning is that the radiance estimate values were very

different in comparison to the emissive values of the light sources (and later caustic radiance estimates). Thus, the radiance estimate values were scaled in order to even the light levels in the image, so that all objects could be seen clearly. Otherwise, the some regions were far too bright. The scaling can be thought of in a similar way to the exposure time of a camera, if the exposure time is too long, the bright areas will overpower the rest of the image.

3.3 Caustics

Whilst caustics can be captured as part of the global illumination map, they are best handled as a special case as they typically require a large amount of photons in order to be rendered well. Jensen (1996) suggests performing an additional photon tracing pass to disperse caustic photons, producing a high resolution photon map.

According the grammar seen in the paper, caustic photons bounce from the light source with one or more specular bounce until reaching a diffuse surface, at which point they are recorded. In order to capture this behaviour, caustic photons can only be reflected specularly or transmitted, and terminate upon hitting a diffuse surface (after being stored).

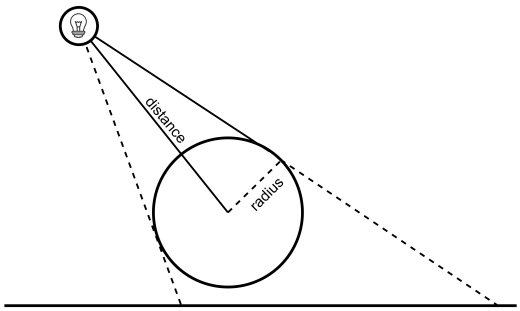

Figure 3.3: Diagram explaining the implementation of the projection map.

Due to the higher concentration of photons required, the regular method of random photon dispersal cannot be used. A projection map was implemented in order to direct the emitted photons towards transmissive objects only. The center point and radius of objects have been pre-determined by the use of a bounding sphere, see Section 4.2.1. This means that a vector can be drawn from the light source to the center point, allowing for the angle between the center and edge of the bounding sphere to be calculated through trigonometry. The angle and vector define a cone in which a random direction can be generated, which is then used as a direction for the caustic photons, see Figure 3.3. This ensures that caustic photons will only be fired towards transmissive objects.

The following code snipped details how projection maps were implemented:

```
1  //for each photon...
2  for (Object *obj : objects) {
3      // if object not transmissive, cannot generate caustic
4      if (obj->material->t == 0) continue;
5
6      // calculate vector to object
7      Vector to_obj = obj->centre - light->position;
8      float distance = to_obj.magnitude();
9      to_obj.normalise();
10
11     //calculate angle of right angle triangle from light to object
12     float cone_angle = atan(obj->radius / distance);
13
14     // use angle to generate vector within the cone from light to object, containing the object
15     Vector direction = Utils::random_direction(to_obj, cone_angle);
16     Ray ray = Ray(light->get_position(), direction);
17
18     // specify caustic photon to change behaviour on types of bounce possible
19     Photon photon = Photon(ray, light->intensity, "caustic");
20     trace_photon(photon, depth, ...);
21 }
22 \\scale photons ...
```

Figure 3.4 shows the result of implementing caustic photon mapping. Notice how the light is focused into regions below the glass spheres.

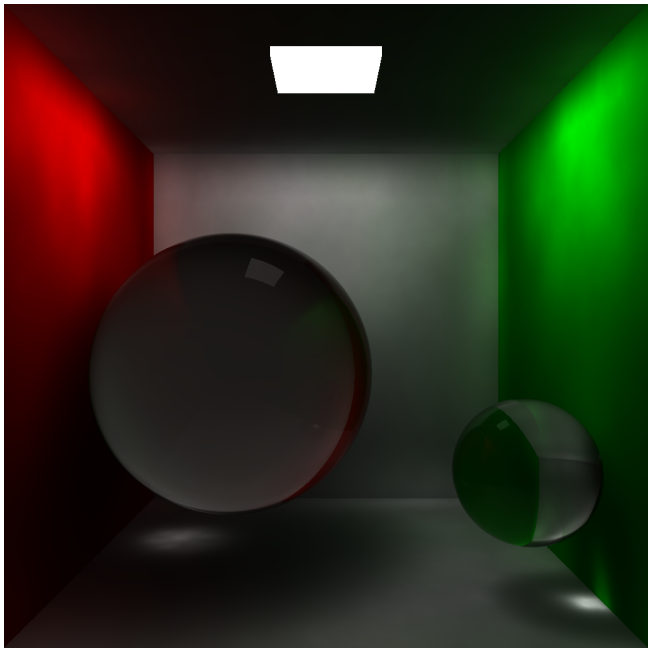

Figure 3.4: Rendered image displaying transmitted caustics through the glass balls.

3.4 Shadow Photons

The rendering step can be optimised through the use of shadow photons. Shadow photons are distributed during the photon tracing step in order to reduce the number of shadow rays generated. At each intersection, the path of the photon is traced further, and at each subsequent intersection a shadow photon is stored, as shown in Figure 3.5.

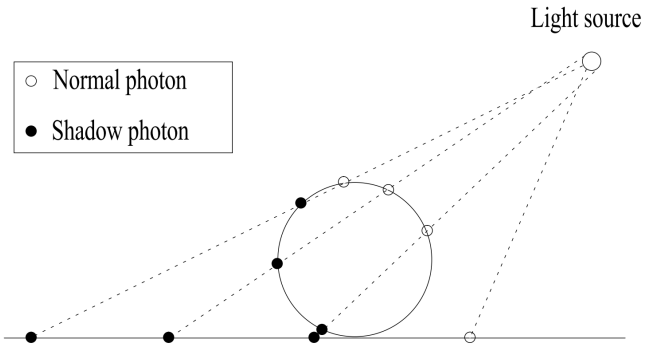

Figure 3.5: Diagram to show how shadow photons are distributed Jensen and Christensen (1995).

The number of shadow photons present at an intersection can then be used to test if shadow rays are required. This was implemented by checking the type of all photons returned when performing a gather, if the majority of photons are shadow photons, then shadow rays should be generated for accurate shadows. However, if not, then the shadow rays are not required.

3.5 Photon Map Serialisation

During development, a large proportion of time was spent generating photon maps prior to each render of the scene. This was found to be very time consuming.

A huge benefit of photon mapping is that the photon map generated is independent of the ray tracing step, so once the map has been created, it can be reused (providing the scene remains unchanged). Thus, in order to speed up development both the KD Tree and the photon objects are serialised and saved to files, allowing them to be opened and read at a later stage.

This optimisation was found to be very effective as no time was spent regenerating photon maps for each render, only loading pre-made photon maps. Another benefit is that because both photon mapping and ray tracing are independent steps, the camera can be moved around the scene without having to redistribute photons. Figures 3.6 and 3.7 were both generated using the same photon map, but with an adjusted camera position.

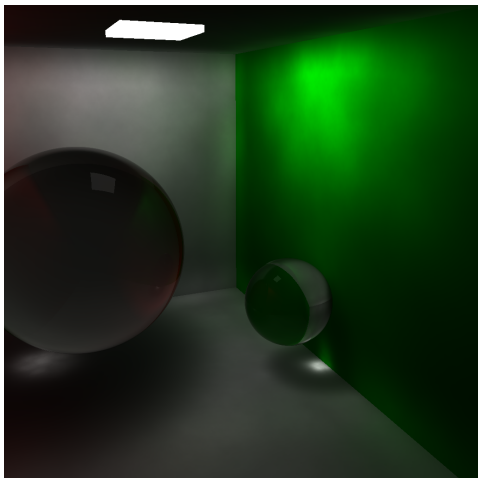

Figure 3.6: Camera positioned in the top left corner of the scene, using a pre-made photon map.

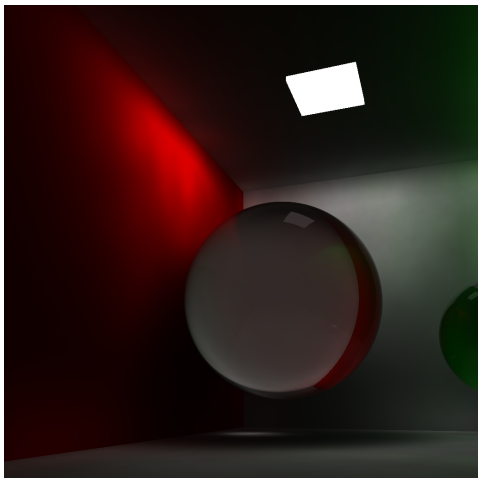

Figure 3.7: Camera positioned in the bottom right corner of the scene, using a pre-made photon map.

Chapter 4

Additional Features

4.1 Procedural Textures

A large benefit of using procedural textures as oppose to regular textures is that procedurally generated textures do not require a source image, so their storage footprint is virtually non-existent. Perlin Noise is a form of structured static that can be used to generate procedural textures through use of pseudo-random numbers (Perlin, 1985). Features of the noise can be parameterised and adjusted depending on the objects desired look and feel of an object.

The technique used to add procedural textures is a C++ implementation of the algorithm described by Perlin (2002). There are 3 main steps to this algorithm:

1. Define a matrix of points where each point has an associated random gradient unit vector.
2. For a given input point, calculate the distance vectors from each corner of the square/cube the point lies, then calculate the dot product of each distance vector with the gradient vector at that corner.
3. Interpolate between the values at the corners of the square/cube and smooth the result using a fade function, as detailed by Perlin (2002).

Generating random gradient vectors could become computationally expensive for large grids, thus the implementation used leverages an array of pre-computed permutations, for which each corner is hashed to retrieve the gradients. The result should produce a texture similar to that seen in Figure 4.1.

The smooth noise texture generated as a result of this algorithm can be sampled at different scales, where the scale determines how much of the noise to "view". Layering and summing the sampled noise across different scales allows for the build up of detailed regions, where the higher zooms contribute more to the overall shape, as shown in Figure 4.2. This process is known as adding turbulence.

[2] `https://gamedev.stackexchange.com/questions/23625/how-do-you-generate-tileable-perlin-noise`
[3] `http://discuss.dynamic-bytes.com/p/11-custom-terrain-using-noise-wip`

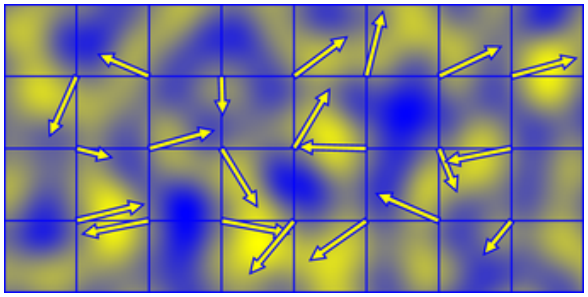

Figure 4.1: Resulant texture through interpolation of values at each point, with example gradient vectors annotated on top[2].

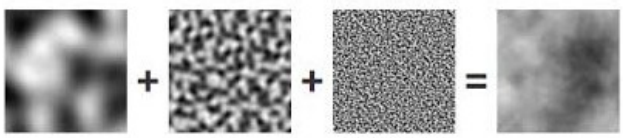

Figure 4.2: Displaying the result of summing perlin noise textures across different[3].

The resulting turbulence texture can then be run through many different functions depending on the desired look and feel of the texture. Many functions exist to model the look of wood and other natural materials. The function chosen for this implementation produces a texture similar to that of marble, as can be seen in Figure 4.3.

The turbulence can be adjusted using various parameters such as the turbulence power, Figures 4.3, 4.4 and 4.5 display how adjusting this parameter affects the "swirls" in the marble.

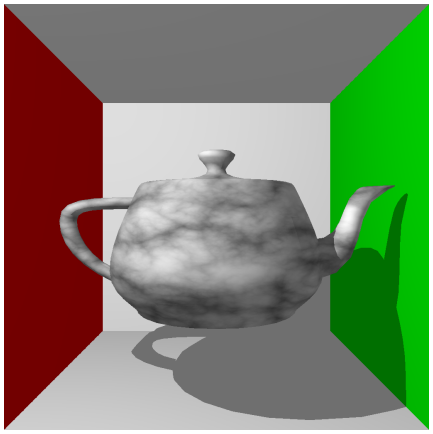

Figure 4.3:
Turbulence Power = 1.

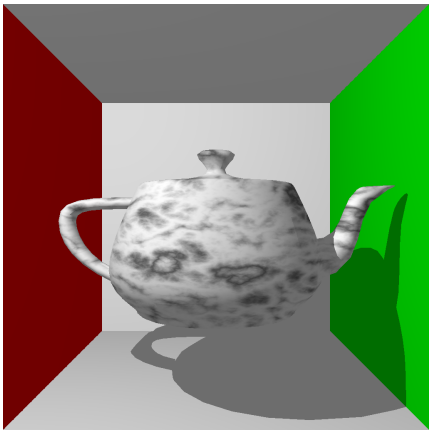

Figure 4.4:
Turbulence Power = 4.

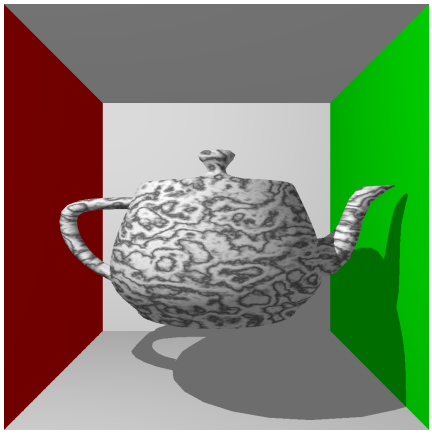

Figure 4.5:
Turbulence Power = 8.

After generating, the texture must then be mapped to an object, known as UV mapping. This is achieved by mapping points of intersection from the surface of the object onto the generated texture, and sampling its colour. When an intersection occurs, the position of the point is scaled based on the maximum

height and width of an object. This calculation allows for the point to be represented as a ratio in relation to the entire size of the object, which can be used to sample a point within the texture. The sampled point is combined with the base colour of the object to create lighter and darker regions.

4.2 Phong Shading

Another additional feature implemented was Phong Shading, a technique used to smooth the appearance of mesh objects (Phong, 1975). In order to apply phong shading, the vertex normals must be calculated, this is achieved finding the resultant vector of all neighbouring face normals. Then, barycentric coordinates can be used to interpolate the normals across the face of the triangle during intersection, as seen in Figure 4.6.

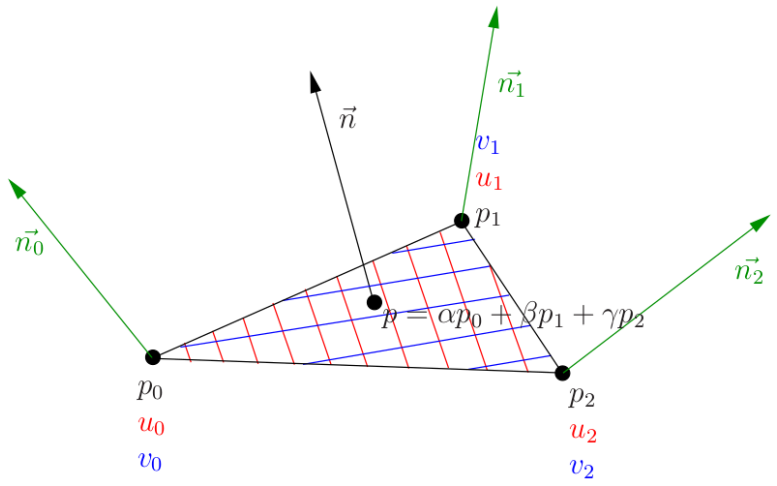

Figure 4.6: Diagram showing how barycentric coordinates can be calculated[4].

The results of implementing phong shading can be seen in Figures 4.7 and 4.8, notice how the spout and lid handle are much smoother.

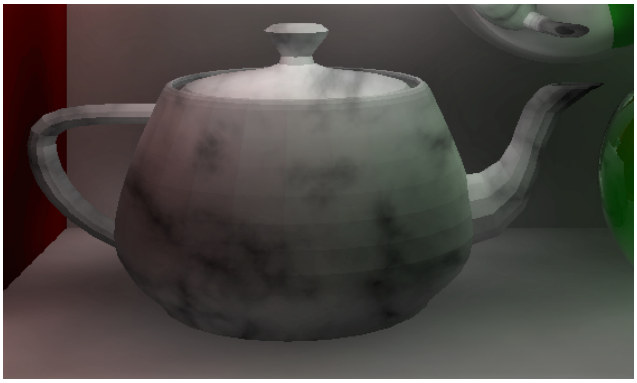

Figure 4.7: Flat shading.

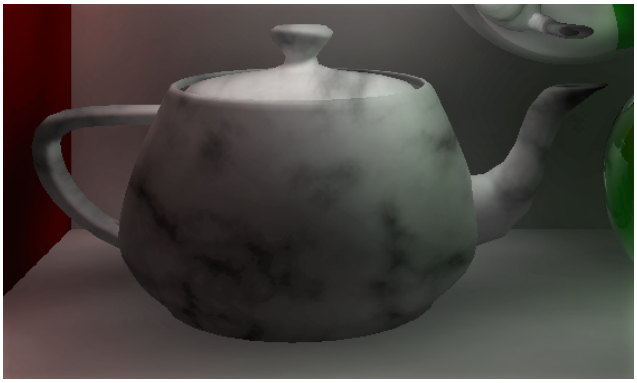

Figure 4.8: Phong shading.

[4] https://github.com/ssloy/tinyrenderer/wiki/Lesson-6bis:-tangent-space-normal-mapping

4.2.1 Bounding Sphere

In order to reduce the rendering time of scenes containing mesh objects, bounding spheres were implemented to encapsulate each object object and act as a single gateway intersection test.

Mesh objects contain many faces, the teapot.ply file contains $\approx$ 2000 faces, all of which must be tested during ray tracing, resulting in slow render times. Whereas spheres allow for fast intersection tests as they are modelled implicitly, and thus intersection tests can be performed very quickly.

Bounding spheres are spheres that encapsulate all points within a mesh object, and act as a first test for intersection. If a given ray does not intersect with the bounding sphere, then the ray will not intersect any of the triangles within the mesh, thus the intersection tests for that object can be skipped. However, if the ray does intersect the sphere, then it is likely it will intersect with at least one of the triangles within the mesh, and so standard intersection test is carried out for the entire mesh.

Jack Ritter's Bounding Sphere algorithm was used for fast bounding sphere approximationRitter (1990). Whilst more efficient algorithms do exist, the given algorithm is much easier to implement and understand. At a high level, the algorithm finds the minimum and maximum points within the point cloud for a given mesh and then finds the point central to those points. The radius can then be determined from the distance between the minimum and maximum points divided by 2. If all points are encapsulated, then the bounding sphere is correct and the radius and center points can be stored and used to create a sphere for the mesh object. In some cases, the sphere may not fit correctly, and so the radius is increased incrementally until all points are contained.

Drastic rendering improvements were seen after the introduction of bounding spheres. Additionally, the improvement scales as the size of the mesh object decreases, as the size bounding sphere will also decrease. Although an upfront cost of creating the bounding sphere is introduced, it reduces the overall rendering time by skipping many unnecessary intersection tests.

4.2.2 Parallelism

Due to the nature of raytracing and its implementation, each ray generated is independent of one another. Thus, ray tracing can be easily parallelised through the use of libraries such as OpenMP. Adding the '-fopenmp' compilation flag and `#pragma omp parallel for` in front of the ray tracing loop, allowed for each pixel colour to be determined simultaneously across many threads.

Unfortunately, due to time constraints, the parallel implementation was not able to be extended to support photon mapping, and has been temporarily disabled.

4.3 Result

Combining all of the techniques and features discussed in this report lead to the generation of the image in Figure 4.9.

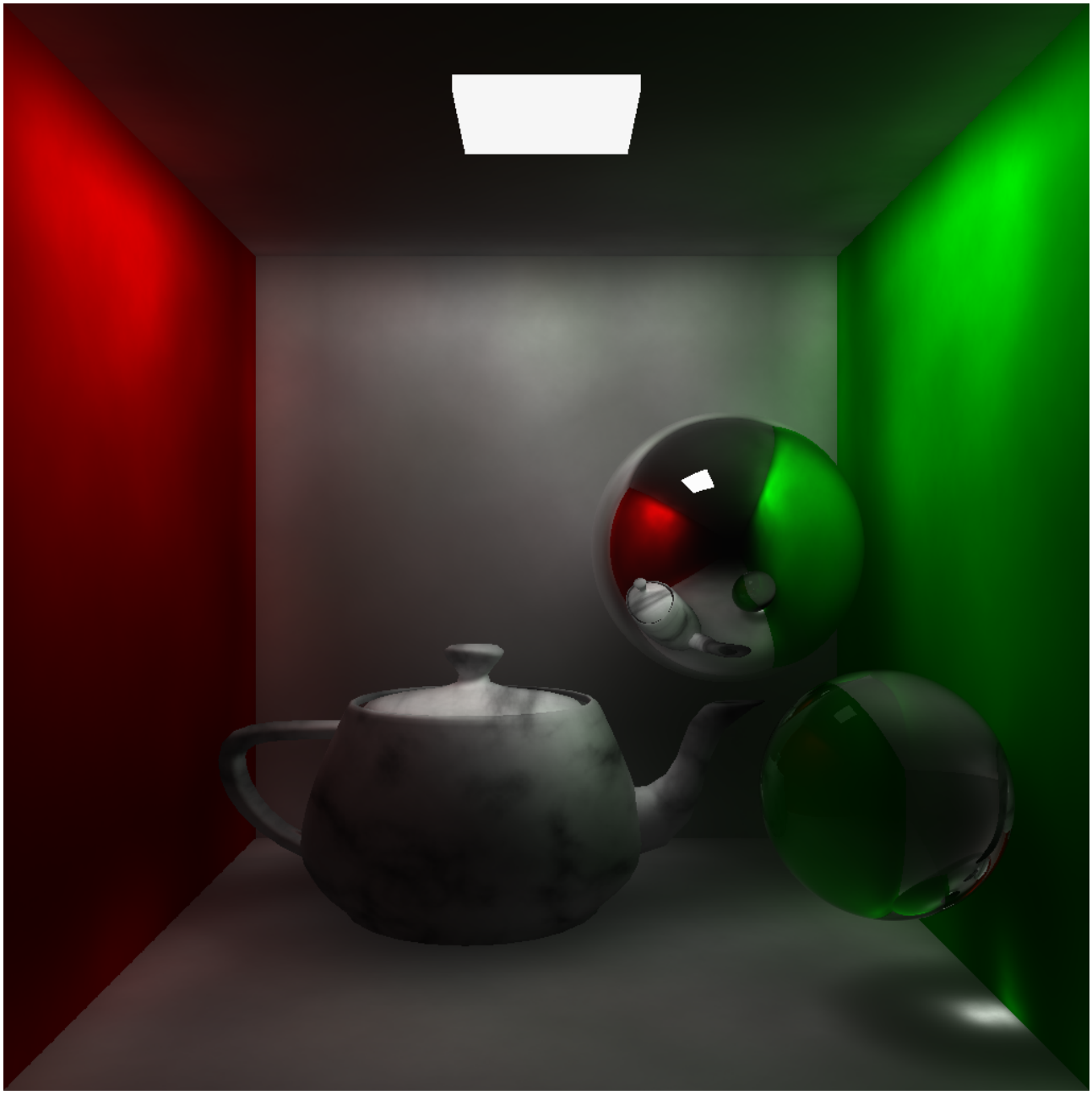

Figure 4.9: Photon mapped image with 50,000 global photons, 10,000 caustic photons per transmissive object with 600 and 200 neighbours respectively.

Chapter 5

Just for fun...

5.1 Utah Teapot in real life

During the development of this ray tracer, a lot of time was spent staring at the Utah Teapot to ensure it was being rendered correctly, what better way to ensure correctness than to compare with a physical model!

Figure 5.1: 3D printed model of the Utah Teapot printed in grey PLA.

Figure 5.2: 3D printed model of the Utah Teapot printed in grey PLA, shown with removable lit.

5.2 Lithophane

The example image generated by the ray tracer in Figure 4.9 was also used to create a Lithophane. Lithophanes are thin pieces of decorative artwork whose thickness vary depending on the amount of light that needs to be let through from a light source located behind the piece. The thinner the region, the more light is let through, representing a brighter area in the image. The lithophane in Figures 5.3 and 5.4 was 3D printed in white PLA plastic and created using 3DP Rocks[1].

[1] http://3dp.rocks/lithophane/

Figure 5.3: Lithophane placed on table in order to view the depth.

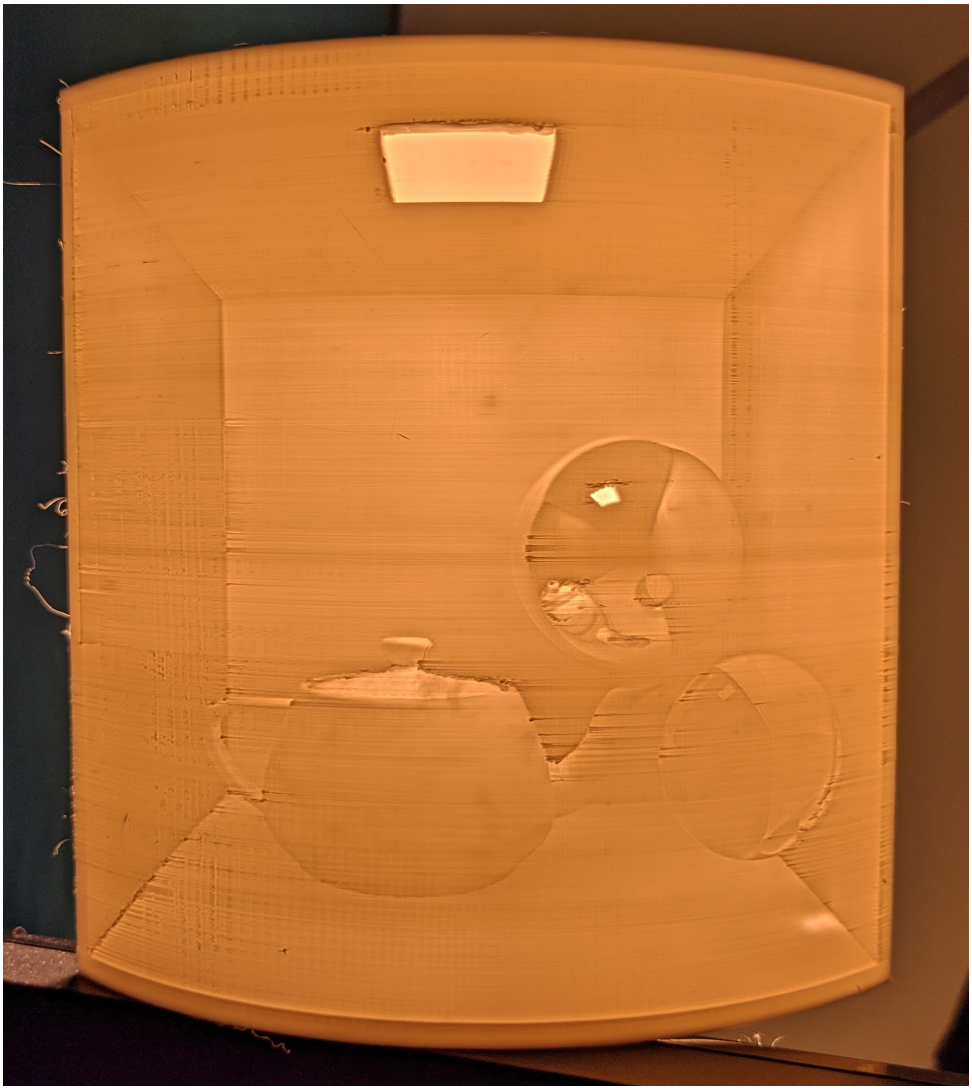

Figure 5.4: Lithophane held up to the light in order to see the effect.

Bibliography

Blinn, J.F., 1977. Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph.* [Online], 11(2), pp.192–198. Available from: https://doi.org/10.1145/965141.563893.

Cameron, K., 2019. Lecture 6 global lighting. p.19.

Dubla, P., 2011. *Interactive global illumination on the cpu*. Ph.D. thesis.

Jensen, H.W., 1996. Global illumination using photon maps. *Rendering techniques' 96*. Springer, pp.21–30.

Jensen, H.W. and Christensen, N.J., 1995. Efficiently rendering shadows using the photon map. *Edugraphics+ compugraphics proceedings*. pp.285–291.

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

Lvovsky, A.I., 2015. Fresnel equations. *Encyclopedia of optical and photonic engineering (print)-five volume set*. CRC Press, pp.1–6.

Perlin, K., 1985. An image synthesizer. *SIGGRAPH Comput. Graph.* [Online], 19(3), pp.287–296. Available from: https://doi.org/10.1145/325165.325247.

Perlin, K., 2002. Improving noise. *ACM Trans. Graph.* [Online], 21(3), pp.681–682. Available from: https://doi.org/10.1145/566654.566636.

Phong, B.T., 1975. Illumination for computer generated pictures. *Commun. ACM* [Online], 18(6), pp.311–317. Available from: https://doi.org/10.1145/360825.360839.

Ritter, J., 1990. Graphics gems [Online]. San Diego, CA, USA: Academic Press Professional, Inc., chap. An Efficient Bounding Sphere, pp.301–303. Available from: http://dl.acm.org/citation.cfm?id=90767.90836.