Advanced Computer Graphics Report

1.0 Note
Regrettably throughout production of this project many of the work in progress pictures captured were corrupted and lost. Where possible I have recreated them, but some (namely those generated during development of photon mapping) were not reproducible without major code changes.

2.0 Past Work
Before starting on this project some previous work was undertaken. While I shall offer a brief overview, understanding of this work is assumed throughout this report. While I was able to complete all labs, I opted to use the starter code provided as the basis of my project as I deemed it to have a more suitable code structure than my initial labs.

2.1 Ray Tracing
Here the teapot was represented by a series of triangular primitives. Simulated rays of light were projected into the scene at each pixel, a triangle intersection algorithm detected whether a primitive had been hit. The depth from the viewer at which this hit occurred was recorded and then used to visualize the teapot, with pixels closest to the viewer appearing black, and pixels further away appearing closer to white. *[see appendix 1]*

2.2 Phong Lighting
Here the Phong illumination model was used to provide convincing lighting. This model breaks lighting down into three core components: Ambient lighting, constant background lighting that is used to model reflected indirect light; diffuse, which shows the effects of light on matte surfaces; and Specular, focused highlights that model light hitting a shiny surface. *[see appendix 2]*

2.3 Shadows
Convincing shadows were added by implementing "shadow rays" these rays operate similarly to visibility rays but are generated on a surface and travel towards a light source. If an object is detected in between the surface and the light source, then the surface is in shadow. *[see appendix 3]*

3.0 Reflection
So far only direct illumination has been considered. In real life light bounces off reflective surfaces, on a shiny surface this allows us to see the surroundings of an object on the surface itself (consider a mirror as the ultimate example of this). The same effect has been modeled here by introducing secondary rays which are generated from an object and are projected out into the scene.

3.1 Implementation
Firstly a Boolean flag "reflective" was added to the material class in order to identify an object as reflective. During ray tracing after a color has been computed for a given surface the reflective flag is checked. If the flag is found, then a new secondary ray is generated.
The position of the new ray is the same position as the surface hit but moved by a tiny amount along the surface normal, negating the effects of any small rounding errors. The direction is calculated using the equation $R = E - 2.0 * (N.E) * N$ provided in lectures. Here E refers to the incoming ray direction and N refers to the normal of the surface that has been hit. Any direction vector must be normalized in order to properly represent the rate of change, as direction vectors of different lengths and directions cannot be usefully compared. As such all directional vectors throughout the code including here are normalized before use.

Once the direction and position are calculated ray tracing is called recursively using this new secondary ray with a new variable "reflectionColour" storing the colour of the reflected ray. To avoid infinite recursion the recursion is halted at a set depth when there is negligible contribution to the image. A level is passed in and is decremented by one on each recursion, once it hits zero recursion is halted. Trial and error was used to settle on a depth value of 4 which gave the best balance of speed and visuals. Finally, the new reflection color is scaled by a reflection coefficient kr (in the case of a non-transparent, reflective surface) and added to the final colour. *[see appendix 4, 5]*

4.0 Refraction

Thus far all objects have been modeled as opaque, now transparency will be introduced. In the real-world light travels differently in different materials, this makes it appear as though light is bending. An easy way to see this is a solid glass ball, which flips its surroundings. This behavior has been simulated within the ray tracer.

4.1 Implementation

Similarly, to reflections, a new material flag was added, "transparent" to indicate a transparent material. Previously reflection was scaled by the kr value associated with a given material, but with the introduction of transparency this was no longer acceptable in all cases. In the real world the amount of reflection and refraction visible (represented by kr and kt) varies depending on the viewing angle. This relationship is modeled by the two Fresnel equations:

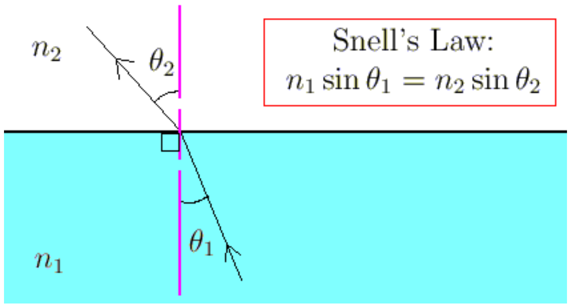
$$F_{R\parallel} = \left(\frac{\eta_2 \cos\theta_1 - \eta_1 \cos\theta_2}{\eta_2 \cos\theta_1 + \eta_1 \cos\theta_2} \right)^2,$$
$$F_{R\perp} = \left(\frac{\eta_1 \cos\theta_2 - \eta_2 \cos\theta_1}{\eta_1 \cos\theta_2 + \eta_2 \cos\theta_1} \right)^2.$$

An average of these equations gives the amount of reflected light that is seen:

$$F_R = \frac{1}{2}(F_{R\parallel} + F_{R\perp}).$$

Scratch Pixel [1] supplied a useful function to calculate a kr term using the incoming ray direction, surfacer normal and refractive index, which I adapted and added in to my code. If the new kr value is < 1 (meaning no internal reflection) and the transparent flag is true then refraction takes place.

When refraction takes place two rays are generated, one representing the path of light through an object, and one representing its path upon exit. For the internal ray direction is calculated using a method derived from Snells law[2] :



This law states that the ratio of the sines of the angle of incidence and angle of refraction is equivalent to the opposite ratio of the indices of refraction.
Standford University[3] shows how to derive the following equations from Snells law.

$$\mathbf{t} = \frac{\eta_1}{\eta_2}\mathbf{i} + \left(\frac{\eta_1}{\eta_2}\cos\theta_i - \sqrt{1-\sin^2\theta_t}\right)\mathbf{n}$$

$$\sin^2\theta_t = \left(\frac{\eta_1}{\eta_2}\right)^2 \sin^2\theta_i = \left(\frac{\eta_1}{\eta_2}\right)^2 (1-\cos^2\theta_i)$$

This method is used to calculate the direction of light upon entering the new material. The position is simply the surface hit moved along the new direction vector slightly to avoid rounding errors. A hit test is then done to see if this internal ray has hit the other side of the object. If this is the case, then an external ray is generated. The direction and position are calculated the same way as previous expect here the refractive indexes are swapped and the hit normal is negated so that it lies on the correct side of the surface. Ray tracing is called recursively using this new external ray with a new variable "refractionColour" storing the colour of the refracted ray.
I initially used recursion to create the internal ray, with a flag attached to each ray labeling it internal or external which changed the refraction calculation accordingly. However, I felt this method was marginally less efficient and proved harder to debug. Finally, the reflected colour is scaled by kr just as before and the refracted colour is scaled by kt (calculated as 1 - kr due to conservation of energy). These are then both added to the final colour. *[see appendix 6,7]*

4.2 Problems Encountered

Refraction was initially tested using the same teapot and sphere scene used in *appendix 5*. As shown in *appendix 8*, refraction appeared to be working accurately however it was not working for other scenes. Debugging this feature initially proved quite challenging, I initially logged the path of every refractive ray but the sheer number of rays generated made finding useful information difficult. To solve this problem, I added a boolean flag "mid" to each ray. This flag was only set to true if the ray was being generated from one of the 4 most central pixels. This allowed me to accurately visualize which ray path my console output was referring to. I kept this method of debugging for subsequent features.
I discovered that the problem was because the refracted rays were bouncing off the inside of objects and being reflected back, rather than exiting the surface on the other side. The reason that the scene in *appendix 8* appeared to be accurate was because the sphere was intersecting the teapot and the light ray was bouncing off the sphere before ever reaching the outer edge of the teapot.

4.3 Future Work

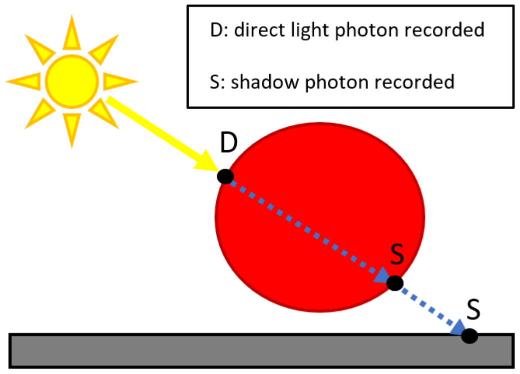
There are a couple of limitations to the transparency model that I implemented. Firstly, objects inside transparent objects are not shown convincingly *[see appendix 9],* and secondly translucency is not possible only perfect transparency. Adding colour to a transparent object goes some way to making an object appear translucent but the effect is not convincing *[see appendix 10].*

5.0 Photon Map

When a simulated photon hits a surface, there will be lots of other similar photons nearby. This can be leveraged to cut down on the number of shadow rays that need to be generated. A two pass algorithm has been implemented that firstly builds a map of sample photons in a KD-tree data structure, and secondly uses this information to infer if an area is in shadow or not.

5.1 Implementation

Before ray tracing occurs the 'buildPhotonMap' function is called. 5,000,000 photons are generated, after some experimentation this number was settled on as it reduced visual artefacts while not slowing the program to unusable levels. For each photon a new ray is generated. A photon map aims to more accurately model the path of light than a normal ray tracer. In real life light sources generally don't point in a single direction but rather emit light uniformly. To this end lights were changed from directional lights to positional lights. Rays are sent in a random direction from a light source. The direction was initially entirely random, but this led to a lot of wasted photons, instead I opted to generate the direction using spherical coordinates to get a more even distribution. The method for this was adapted from a stack overflow answer but without the radius component [4]. A new flag is added to the ray class: 'shadow', this is set to false by default. The ray is then passed into a 'raytracePhoton' function. If the photon ray hits an object it is recorded in the photon map along with the position at which it hits. After this the shadow flag is set to true and the ray is continued, any other points that the ray hits are then recorded in the photon map. As the ray has already hit an object, any other points that it hits must be in shadow.



Photons are recorded in a 3-dimensional K-D tree data structure which has been implemented based on the version outlined by geeks-for-geeks [5]. A K-D tree is a space partitioning data structure, it is ideal for the photon map as it allows for fast retrieval of photons given a nearby point in x,y,z space.
Once the photon map has been populated ray tracing takes place as normal. Once a ray hits a surface the photon map is consulted and a sample of the 50 closest photons is taken. The number of shadow rays and direct light rays are calculated. If the photons are all shadow rays then it is inferred that the surface is not lit at the point of ray intersection, if the photons are all direct then it is inferred the surface is lit. Only if there is a mix of shadow and direct photons are shadow rays generated. By only generating shadow rays when ambiguous the number of rays generated is vastly decreased. *[see figure 11]*

5.2 Problems Encountered

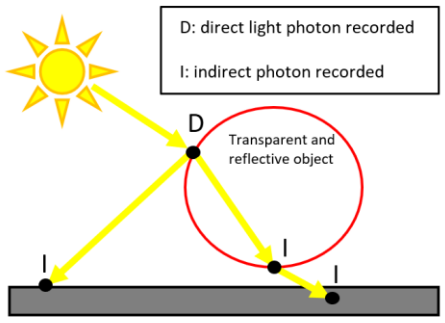
Initially the during construction of the photon map, shadow rays were only generated on the external edge of an object to avoid shadows being recorded on internal surfaces. However, after consideration I realized that it didn't matter as shadow photons would need to be recorded on the opposite side of objects (consider a shadow on the side of a ball furthest from a light source) and the photons would be treated identically whether they were recorded internally or externally as they would for all intents and purposes have the same position.In some instances, during testing, the photon map wouldn't populate if there weren't enough objects in the scene, this would cause crashes. To prevent this a boolean return was added to the kd-tree to ensure that the photon map is only accessed if it exits.

6.0 **Caustic Photon Map**

Caustic light is any light that has been reflected or refracted. Recording caustic light in a separate photon map can help us to model light more accurately, showing refracted light paths emitted from transparent objects. Additionally, it is used to eliminate the ambient term from the lighting model. In the real-world ambient light is caused by indirect light which bounces around to reach places that are not directly lit. The ambient term is an abstraction of this, but the photon map allows for a more accurate representation. Ray tracing could be used to model this behavior but there would be far too many photons, this method builds a high-fidelity map and uses it to make informed estimates of the colors of surfaces.

6.1 Implementation

For each photon, after the first photon map is added to, a second ray is produced with random direction starting at the light source. Two new flags were added 'direct' and 'indirect'. Direct meaning any light that has come directly from a light source, and indirect being light that has been reflected or traveled through a surface


D: direct light photon recorded
I: indirect photon recorded
D
Transparent and reflective object
I
I
I

When a photon hits a surface there are 3 possibilities depending on the surface, it could be absorbed by the material, it could be reflected, or it could be transmitted through the surface. In any of these three cases the photon is recorded in the caustic photon map. The outcome is decided through Russian roulette. Each material is given a pr and pt value denoting the probability to transmit or reflect light, pr is the same as the kr of a material. pr + pt cannot exceed 1 and 1 – pt – pr denotes the probability that light is absorbed. Using these probabilities an outcome is randomly selected for the photon. If the photon is reflected a random direction which reflects out into the scene is chosen. If the photon is transmitted it is refracted using the same method detailed previously. Once the photon is absorbed tracing is finished.
Within the second pass rendering stage the caustic map is accessed after the shadow calculations detailed section5.1 take place. The closest 50 caustic photons to a hit are considered. The colour of each photon is scaled by 1 – distance from the hit / the distance of the furthest away photon in the sample. This adds greater value to the photons closest to the hit as these are more likely to be of the same colour. These scaled photon colours are all added together and divided by the total number of photons to produce a scaled average of indirect light.

During final colour calculations direct, indirect, reflected, and refracted light are all added together to decide the final pixel colour.
The previous ray tracing method light transport equation was L(D)(S)*E, where L = light, E = the eye, D = diffuse and S = specular. This method gets us much closer to the real life light transport equation: L(D|S) * E. *[see appendix 12]*

6.2 Problems Encountered

Initially I didn't scale caustic photon colour. This resulted in muddy images *[see appendix 13].* I made the scene render only indirect color so I could get a better idea of what was happening. After changing the scale to favor nearby photons the indirect contribution was much more useful *[see appendix 14].*

6.3 Future Work

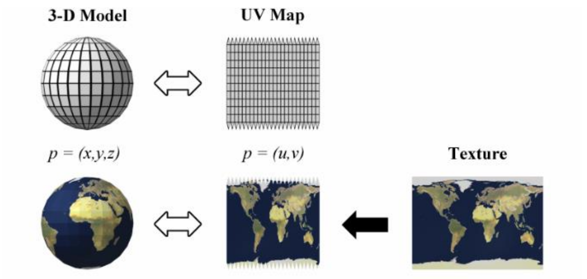
Due to the random nature of reflected photons, many of them are wasted, using a BRDF to constrain the reflection direction would prevent this. While some caustic effects are visible from transparent objects, to get caustic light working optimally a high number of photons need to be directed at transparent objects specifically, unfortunately this was not implemented due to time constraints.

7.0 Spherical texture mapping

I decided that as part of my final scene I wanted a planet in the sky, to achieve this I would need to map a texture onto a sphere. I adapted a method found in Ray Tracing: the Next Week as explaining in Vincent Lis blog [6].

7.1 **Implementation**

As I only wanted one planet this implementation only supports a single texture for simplicity. A flag 'usetexture' is added to the material class, this dictates whether a texture is to be projected. The texture is loaded using the public domain stbi image loader [7].



At the end of the rendering stage, the x,y,z coordinates of the hit position on a sphere are converted to normalized U,V coordinates from spherical coordinates:
ϕ=atan2(y,x)ϕ=atan2(y,x)
θ=asin(z)
u=$\phi/2\pi$u=$\phi/2\pi$
v=θ/π
These new U,V coordinates are mapped to a point on the texture and used to return the correct RGB colour value, normalized to be between 0 and 1. The final calculated pixel colour is then scaled by the texture colour to retain lighting effects. *[see appendix 15]*

7.2 Problems Encountered

Simply adding the texture colour rather than scaling was trialed but this overpowered all other colours in the scene. It in essence added more light to the scene without an accompanying light source. *[see appendix 16]*

References

[1] Scratchapixel. (2014, August 15). Retrieved December 29, 2020, from https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-shading/reflection-refraction-fresnel

[2] (n.d.). Retrieved December 29, 2020, from https://www.math.ubc.ca/~cass/courses/m309-01a/chu/Fundamentals/snell.htm

[3] De Greve, B. (2006, November 16). Reflections and Refractions in Ray Tracing. Retrieved December 29, 2020, from https://graphics.stanford.edu/courses/cs148-10-summer/docs/2006--degreve--reflection_refraction

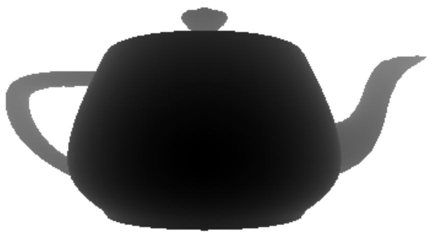
[4] Tim McJiltonTim McJilton 5 (1960, April 01). Sampling uniformly distributed random points inside a spherical volume. Retrieved December 29, 2020, from https://stackoverflow.com/questions/5408276/sampling-uniformly-distributed-random-points-inside-a-spherical-volume

[5] K Dimensional Tree: Set 1 (Search and Insert). (2020, November 03). Retrieved December 29, 2020, from https://www.geeksforgeeks.org/k-dimensional-tree/

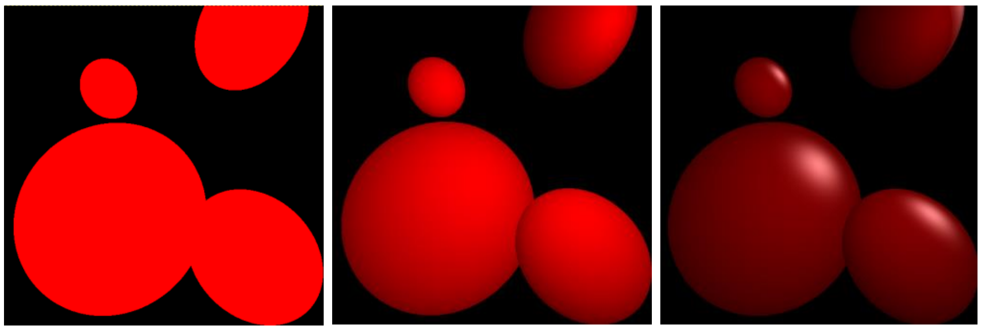
[6] Li, V. (n.d.). Raytracing - UV Mapping and Texturing. Retrieved December 29, 2020, from http://viclw17.github.io/2019/04/12/raytracing-uv-mapping-and-texturing/

[7] Nothings, Git hub user. (2020, July 13). Stb image loader. Retrieved December 29, 2020, from https://github.com/nothings/stb/blob/master/stb_image.h

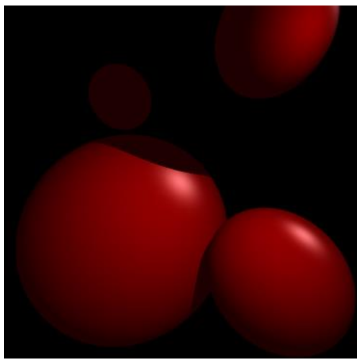
Image appendix



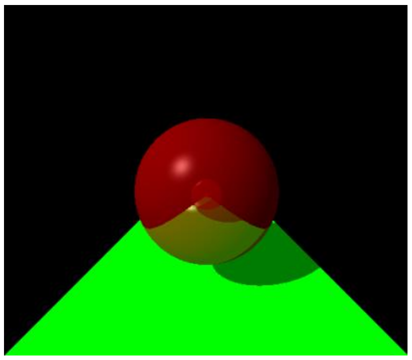
Appendix 1: A first render of the teapot using ray tracing to plot depth



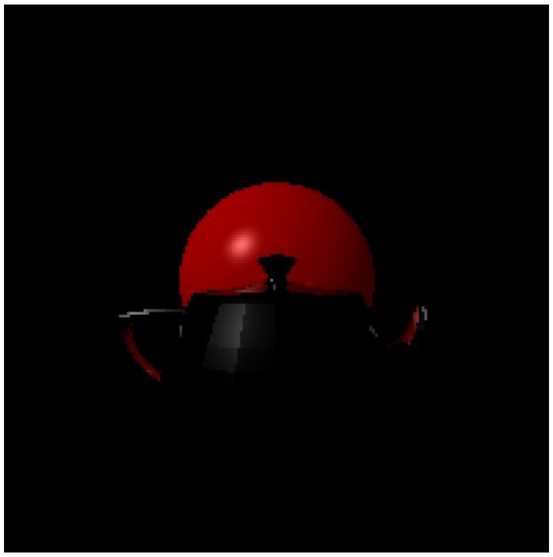
Appendix 2: Ambient, Diffuse and Specular lighting added to a series of spheres

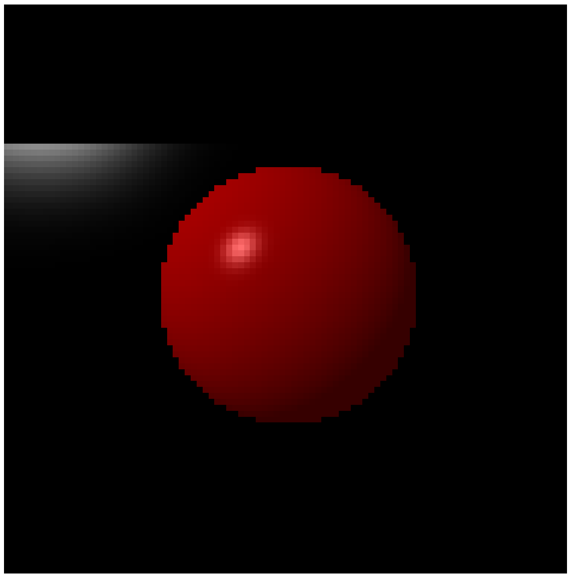


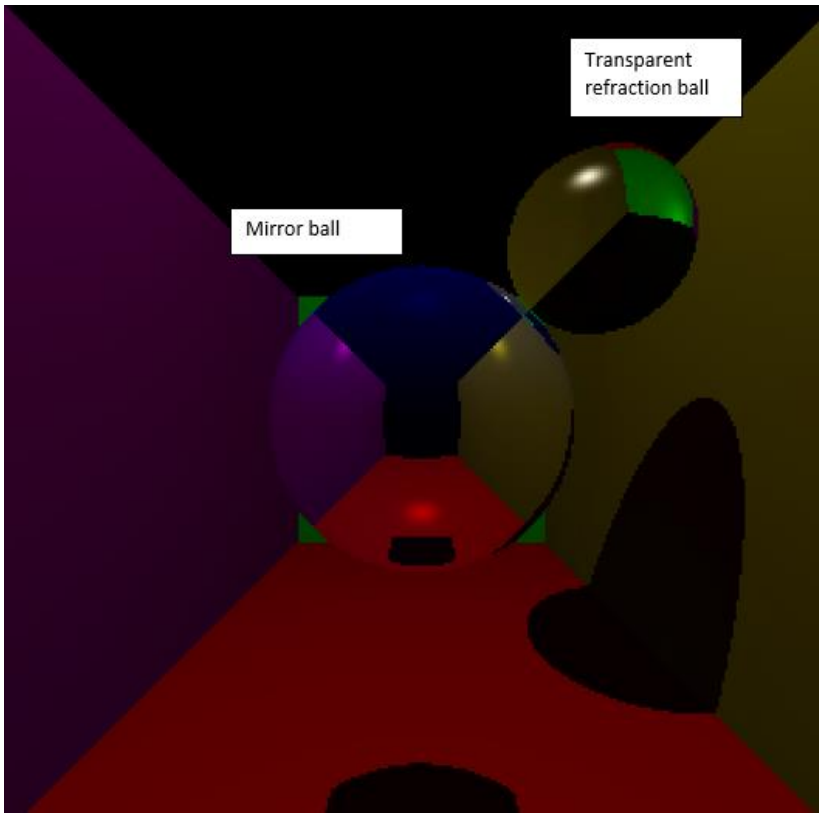
Appendix 3: Shadows added to the previous scene



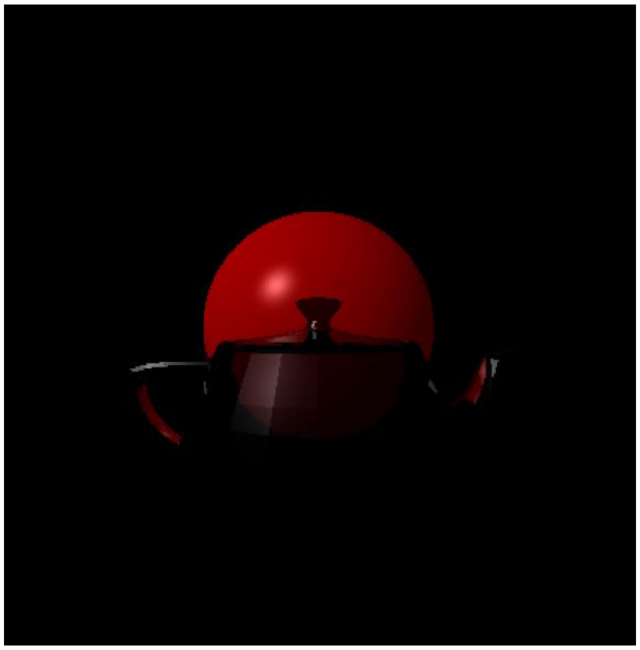
Appendix 4: A red ball reflecting another red ball on the opposite side of a rectangular green plane.


Appendix 5: Reflection applied to the teapot to test reflection on more complex shapes.

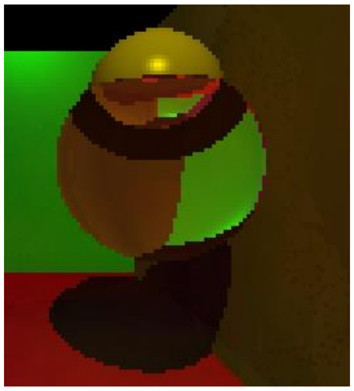

Appendix 6: A thin transparent cuboid in front of a red sphere

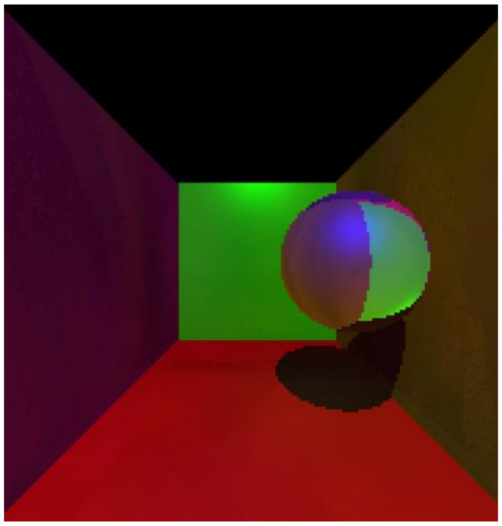


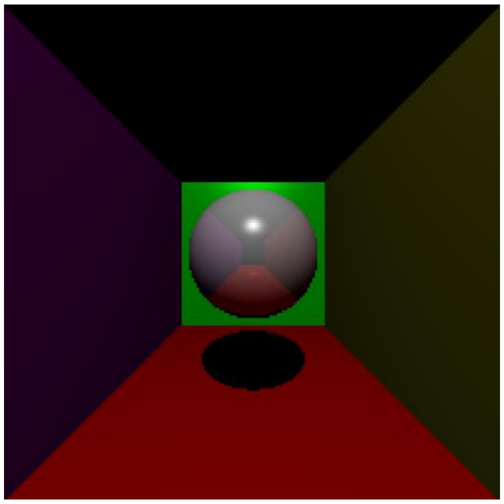
Appendix 7: A transparent and completely reflective ball inside a partial cube

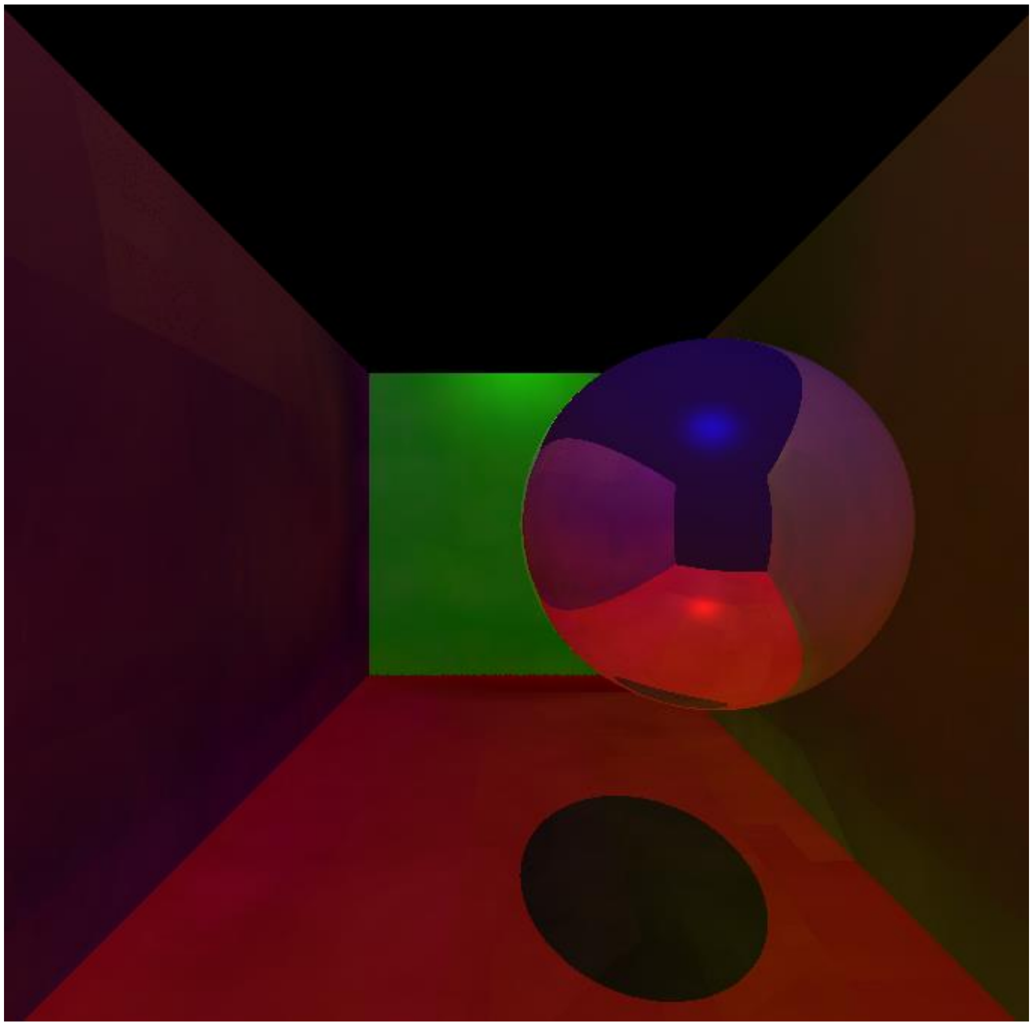


Appendix 8: A sphere seemingly refracted by a transparent teapot

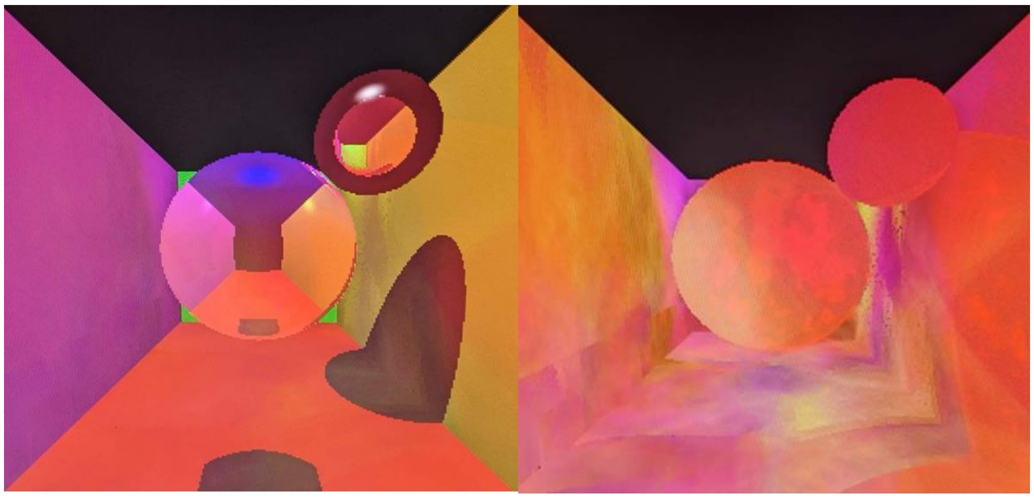

Appendix 9: A yellow ball intersecting with a transparent ball


Appendix 10: A blue transparent ball

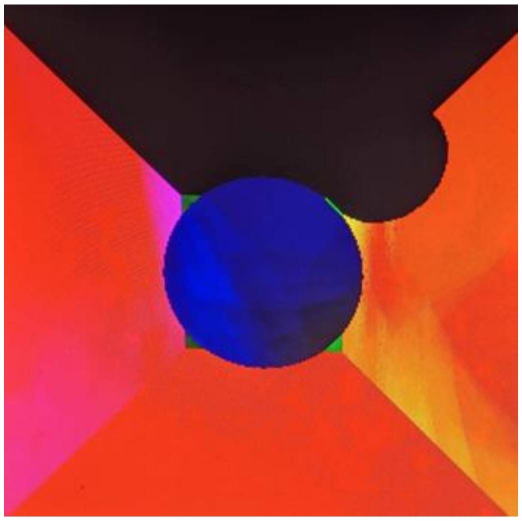

Appendix 11: a simple scene rendered using the photon map, showing shadows still work as intended

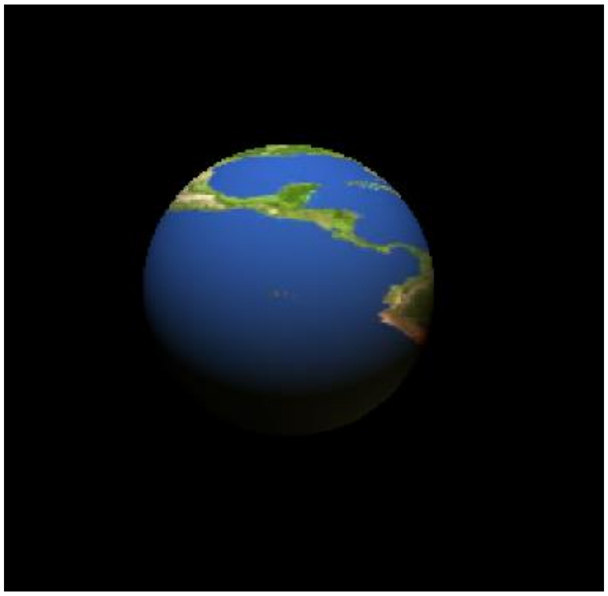


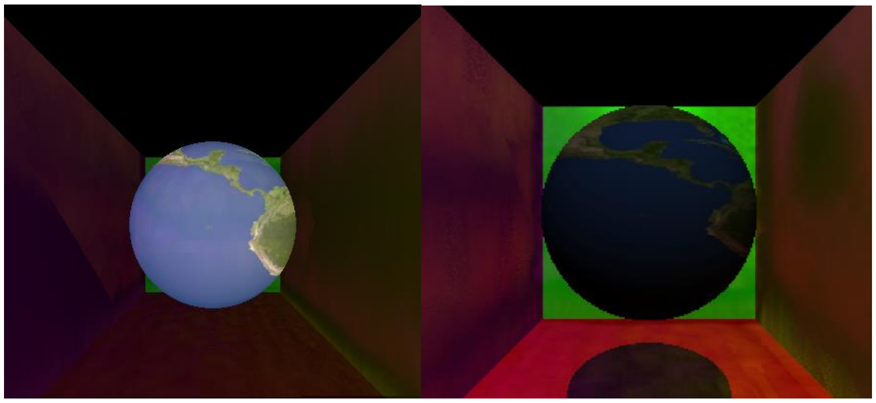
Appendix 12: A simple scene utilizing the caustic photon map showing ambient replacement in the shadow and reflected colour from the walls.

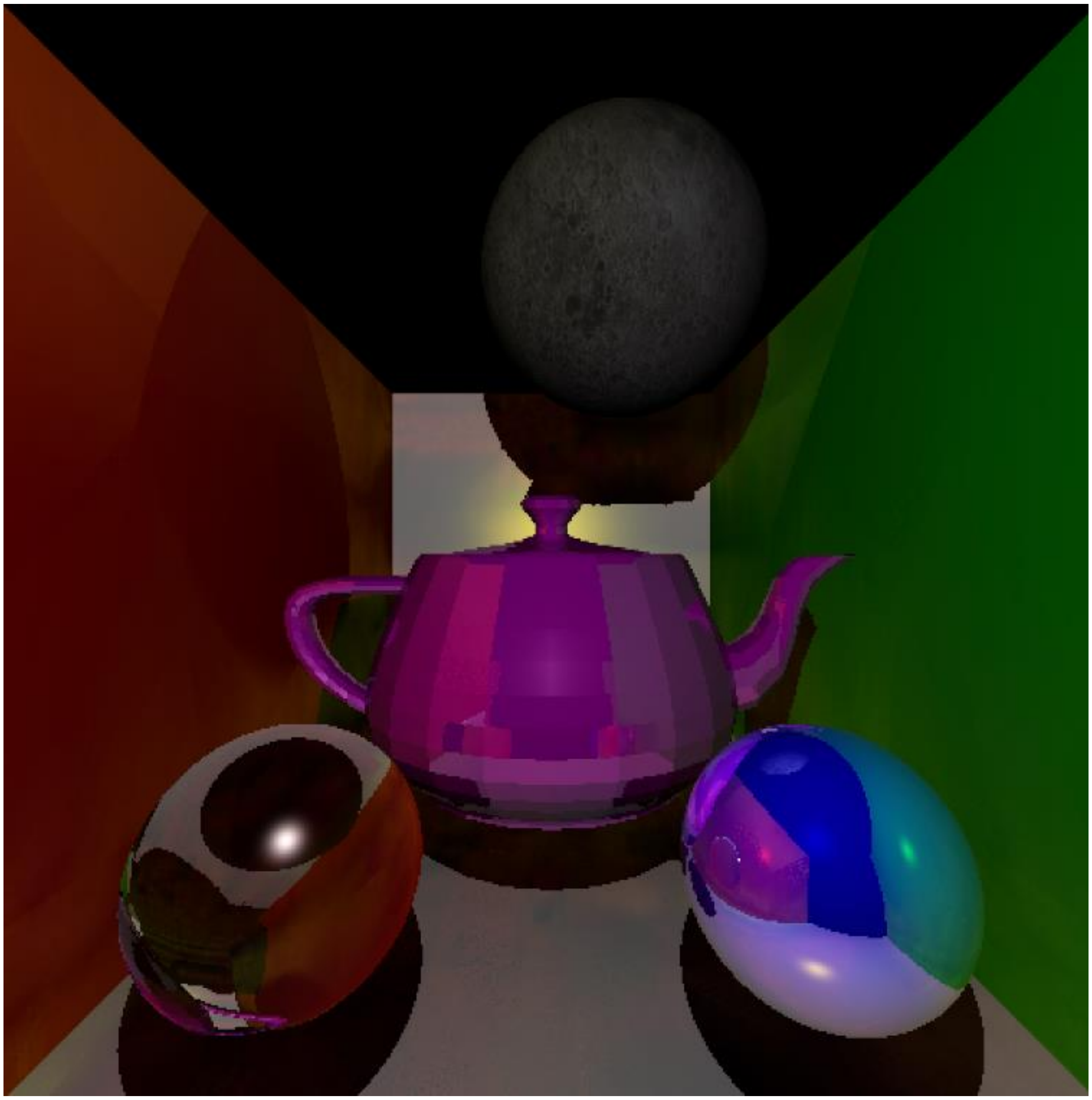


Appendix 13: Indirect contribution before scaling.


Appendix 14: Indirect contribution after scaling.


Appendix 15: A globe rendered using spherical texture loader.


Appendix 16: globe rendered by adding texture contribution and scaling by texture contribution.



Appendix 17: A scene demonstrating reflection, refraction, photon mapping and textures.