Advanced Computer Graphics

Oliver Eisenberg

December 13, 2019

Contents

1 Introduction

Raytracing is a rendering technique that determines the visibly and colour of objects in a scene. The technique revolves around tracing rays, from the eye or camera, through each pixel and calculating the colour upon intersection with the closest object. The principles of raytracing can be extended to visualise shadows, reflection and transmission.

1.1 Visibility

1.1.1 Rays

Rays are defined parametrically to take advantage of the t parameter, this makes determining the object behind and closest to the camera straight forward - as a negative value means that the intersection occurred behind the camera.

1.1.2 Ray Intersections

To be able to add an object to a scene, it must support intersection tests with an arbitrary ray. As more complex objects, like a polymesh, use a series of smaller objects covering basic shapes, supporting these is enough to cover a majority of the meshes. The initial coursework code had sphere intersection support and was therefore expanded to include triangles and planes.

1.1.3 The Camera

The camera model allows the scene to be rendered around the viewer and is used to orientate the scene and emit rays. To be able to do this the camera requires knowledge of its position, where the camera is pointing, the up direction and distance to the image plane.

1.2 Colour and Shading

To determine the colour of the pixel that the ray was traced through, the closest object has to be identified. Therefore, all objects in the scene have to be iterated through before selecting the object. Furthermore the object has to be checked to see if is in shadow, this is done by emitting a shadow ray towards each light in the scene. Intersections found here, mean that the object is in shadow and, for that light source, the colour doesn't have to be calculated.

1.2.1 Local lighting

Local lighting is the illumination of objects direct from the light source(s) in the scene. Objects can reflect lights either diffusly, shown in equation 1, which gives a matt effect, or specular - shown in equation 2, which acts like a mirror.

$$I = I_l k_d (N.L) \tag{1}$$

$$R = I – 2.0 \times (I.N) \times N \tag{2}$$

As surfaces are not perfectly smooth the Phong approximation 3 was used, this gives an appropriate shine to an object.

$$I = I_l k_S (R.V)^n \tag{3}$$

Therefore the total colour calculation, per colour channel, for an ray to object intersection is:

$$I = I_a k_a + \sum I_{ln} (k_d (N.L) + k_s (R.V)^n) \tag{4}$$

Where,

- $I_a k_a$ is the ambinent intensity and coefficent values
- $k_d (N.L)$ is the diffuse coefficent, hit normal and incident light direction values
- $k_s (R.V)^n$ is the specular coefficent multiplied by the reflected ray and viewer direction to the power of n, the distribution controller whoes trypical values are around 20-200.

1.2.2 Self Shadows

When computing shadows self-shadowing can occur. This is caused by rounding errors, where, the shadow ray is generated at a starting point from within the original intersecting object. This causes the shadow ray to hit the inner object and declare that pixel as in shadow - even if it may not be. To resolve this issue the new shadow ray is shifted along the direction of the ray by a small amount.

Testing all objects for shadows can be optimised as, for a single light source, once you found a shadow identifying another doesn't further shadow the pixel. Therefore the loop that checks for shadow can stop at the first occurrence. Further optimisations can be done through the use of a shadow cache. Shadow caching works due to ray coherence where if a ray is being emitted close by to a previously true shadow ray you can assume that ray also in shadow, preventing checking for an intersection.

1.2.3 Global lighting

Global lighting is the addition of secondary rays to compute mirror and transmissive surfaces.

Reflection works by assuming a specular surface is a perfect mirror. Performing the equation 2 and raytracing the result calculates the reflected colour. This is then added to the original colour calculated prior to reflecting. The cycle of ray generation for secondary rays leads to recursion.

Similarly to reflection, transparency lends to recursion. This is because secondary rays are produced to simulate the movement of flight through a transparent object. To calculate refraction, the index of refraction needs to be set on object initialisation, this constant is used to determine how much the light is slowed by when entering the object. Depending on the incident angle and medium the refracting ray will either slow down and bend towards the normal, speed up and bend away from the normal or undergo total internal reflection. This, however, isn't realistic transparency. This is because, for real materials the angle of incidence changes the amount of reflection and refraction that will occur. To account for this the Fresnel equations (7 to 8) are applied. These result in more realistic behaviour.

$$r_{par} = \frac{(\eta cos\theta_i - cos\theta_t)}{(\eta cos\theta_i + cos\theta_t)} \tag{5}$$

$$r_{per} = \frac{(cos\theta_i - \eta cos\theta_t)}{(cos\theta_i + \eta cos\theta_t)} \tag{6}$$

$$k_r = \frac{1}{2} \times \frac{(r_{par}^2)}{(r_{per}^2)} \tag{7}$$

$$k_t = 1 - k_r \tag{8}$$

2 Raytracing Implementation

The completion of lab sheets allowed to get familiar with C++ and to start building practical knowledge on raytracing.

2.1 Labs 1 & 2

Lab one and two allowed to build supporting functionality to use and test the implementation of raytracing. To be able to construct a wireframe, the line drawing function was updated to use an integer based approach using the Bresenhams Line Algorithm. The algorithm is an efficient method used to draw a straight line and avoids the use of floating point multiplication to draw pixels. Instead, Bresenham's algorithm increments axes so that the gradient of the currently drawn line remains similar to the goal gradient, where the goal is the initial vector from point A to B.

Lab two allowed the algorithm to be used to draw wireframes of an imported dataset. The given code in *polymesh.cpp* was updated to be able to read 3D objects from *.ply* files. There are several mesh data structures that can be used to store polymesh objects, the method supported in the *polymesh.cpp* is the Shared Vertex(OBJ, OFF) where it contains an indexed list of vertices and faces (vertex indices). The teapot dataset is comprised of three header lines, these identify the number of vertices and faces that are contained and are iterated through to populate their respective vectors. The shared vertex data structure is adequate, however, more complex structures have the potential of reducing construction, query and improving efficiency. For example the *.ply* file could have used Face-Based, which lacks in edge information, or Edge-Based connectivity, which excludes edge orientation information.

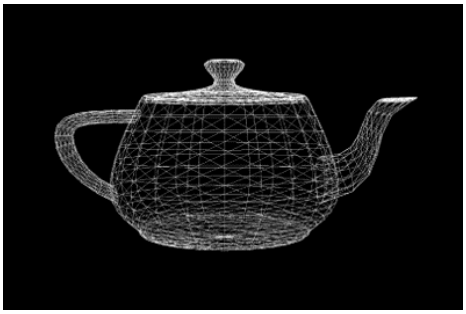


Figure 1: End product of lab two

2.2 Lab 3 - Simple Raytracing

2.2.1 Framebuffer

To be able to test triangle intersections could work through ray intersection tests, the rays were first tested on a depth buffer. The depth buffer uses the paramatric t value in order to visualise the depth of the hit. By suppling the sphere intersection code this could be tested first here before developing and testing triangle intersections. The intersection testing for the sphere was straight forward using psudocode provided from the slides.

2.2.2 Triangle intersection

To compute triangle intersections the Möller–Trumbore algorithm was used. This was used instead of the method on the slides [1] anticipating the requirements for the barycentric coordinates to complete Gouraud shading further in the coursework. Baracentric coordinates allow for Gouraud shading as the values can be used to determine where on the triangle the intersection took place. Being able to follow the general intersection layout for spheres laid out the structure

well for testing the triangles. The end result (figure 2) is shown, where darker surfaces indicate closer proximity to the camera.

Figure 2: End product of lab three

2.3 Lab 4 - Basic Lighting and Shadows

Starting to work on lighting prompted further refactoring. In doing so, a scene class was created to create and store all the objects and lighting for the render. All lights belong to the base light class, this allows to predetermine shared variables and functions a light should have and for lights to be stored in a single vector.

2.3.1 Types of light

2.3.2 Spot Lights - Directional lights

Spotlights were first implemented due to their simplicity. Spotlights are considered to infinitely far from the scene and therefore have no position and all light rays are parallel.

2.3.3 Point Lights

The slides refer to two methods to create point lights, with and without an associated direction. As spot lights are directional, point Lights with a constant intensity were implemented. Allows a point light to be placed between objects to cast shadow outwards. This is demonstrated in figure 3.

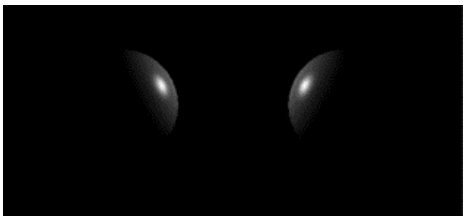


Figure 3: Simple render to illistrate pointlights

2.3.4 Shadows

Shadows are checked by sending a ray from the hit position to each light in the scene. If an intersection occurs the intensity from that light is removed. This allows for shadows of different intensity depending on the source. Originally the program would remove light intensity relative to the number of total lights - disregarding the light source's intensity. This was rectified and updated before the lab 4 submission date.

Figure 4: Teapot casting two shadows

In addition the shadow ray's length must be compared with the distance to the light, this is important as when adding planes to the scene a light source in between two planes can cause a shadow as if one of the planes obstructed the light. Instead the light ray is passing the source, therefore there would be no shadow, but then intersects with the secondary plane.

3 Minor Optimisations

3.1 Colours & Materials

Originally objects had three intensity values representing the three primary colours in addition to the diffuse and specular coefficients. This was later updated to use a material and colour class. The colour class allowed for a collection of operations to be enabled, this was useful for adding and performing operations on colour. This change prompted the addition of materials now, instead of later in the coursework for textures to be applied. Materials were changed to hold specular, transmission values as well as the object's base colour. This simplified the initialisation of objects in my scene class to creating several materials for objects to use.

Furthermore, by splitting intensity into colours it allows the addition of coloured lights and control over what coloured objects can reflect this could also benefit possible diffraction and iridescence rendering.

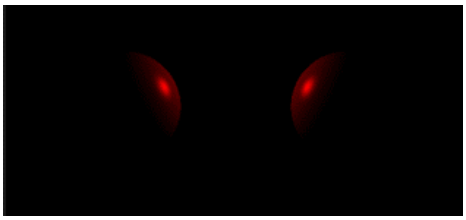


Figure 5: Simple render like figure 3 but with a red light

4 Photon Mapping

Photon mapping is a two pass algorithm that allows to compute global illumination, caustics and precipitation media, building on raytracing. In the real world most light is indirect, a standard raytracer cannot simulate interreflection between diffuse surfaces. The combination of BRDFs and the Rendering Equation provide a way to model photon paths in the correct direction.

4.1 The rendering equation

$$L(x', \omega) = E(x', \omega') + \int \rho'_x(\omega, \omega') L(x, \omega) G(x, x') V(x, x') dA \tag{9}$$

The equation comprises of five terms, unless the object is not emissive. If the object is emissive then $E(x', \omega')$ is the emitted radiance from a point, otherwise it is 0. The other four terms describe the how the radiance of other objects in the scene contribute at the point.

1. $\rho'_x(\omega, \omega')$ describes the contribution from the reflectivity (BRDF) from the other object.
2. $L(x, \omega)$ is the radiance from the other object.
3. $G(x, x')$ is the geometric relationship between the two surfaces.
4. $V(x, x')$ is the visibility of the other object. The equation should only apply to the closest object, therefore it resolves to 0 if the object is obstructed.

This models other techniques like raytracing and radiosity but doesn't support all natural lighting features, for example refraction and caustics.

4.2 BRDFs

Bi-directional Refectance Diribution Functions are a method to specify the amount of light that leaves in relation to the light that arrived. The ambient, diffuse and specular models used when raytracing fit the functions. BRDFs are four dimensional and more parameters can allow for more complex materials. For example adding wavelengths will allows to model Iridescence.

4.3 Two pass algorithm

4.3.1 Pass one

In the first pass, two photon maps are generated with different purposes. One is for high resolution caustic photons and a lower resolution general one [2]. Similarly to raytracing photons are traced through the scene and for each photon hit, type, directional, positional and colour information is stored and, if a secondary photon can be generated from reflection follow and recurse. This process is repeated for all n photons. The photon information is stored in a balanced kd-tree, which is a special case of a binary search tree.

When a photon does hit a surface Russian roulette is used [3] to determine what outcome to choose. The probability for each outcome is calculated using the hit object's diffuse, specular and transmissive values.

```
r = generate_random[0, 1];
if (0 <= r && r < diffprob)
// diffuse reflection
if (r < diffprob+specprob)
// specular reflection
if (r < diffprob+specprob+transmissiveprob)
// transmissive calculation
if (r < 1)
// absorb
```

A Caustic map is created by emitting photons towards specular surfaces only, these photons are stored when they hit a diffuse surface and can be rendered directly as a radiance estimate.

4.3.2 Pass two

Rendering is done using raytracing but with the knowledge from the photon information gathered. Accurate or approximate calculations can be done at each intersection. The accurate method is the normal raytracing method and is used for direct illumination and specular reflections. The approximate uses the radiance estimate is used for everything else.

1. Direct : Either accurately or approximately
2. Specular : Normal Monte Carlo raytracing
3. Caustic : Render caustic photon map directly
4. Soft indirect : approximate is included in the global photon map, accurate can be done using importance sampling

5 Photon Tracing Implementation

Some difficulty was encountered when choosing a library that supported kd trees. This was because they were often static libraries which made integration harder

on a windows subsystem for Linux. This heavily influenced the chosen library to handle KD Trees. The library chosen *Alglib* [4] was created to be added to code like your own classes where you include the relevant headers and compile the *.cpp* files yourself. The library's documentation was enough to create a local wrapper class that allowed easy access when storing and accessing photon from their maps.

5.0.1 Random emission - Lighting

Lights had to updated with relevant random emission direction and position functions. My point light was therefore changed so that all random points started from the point light's position and a random direction could be generated. This behaviour could not be generalised as for spot light the points would have to randomly start in different positions but have the same direction vector.

5.0.2 Specular & Gloss

This was straight forward to implement as it is rendered using standard raytracing techniques that were developed during the lab hand-ins.

5.0.3 Caustics

Caustics are computationally intensive to calculate using standard raytracing methods but was straight forward to implement using a separate caustic photon map. In a caustic map photons are emitted from the light source to reflective and transmissive objects only, this was done by tracing photons to random positions to supportive objects. Later when rendering, the map is calculated using a radiance estimate. A caustic map can be filtered prior to rendering to smooth out hard edges, however the caustic renders I was generated never seemed to be extremely hash - this may have been due to a high nearest neighbour count in my caustic photon map.

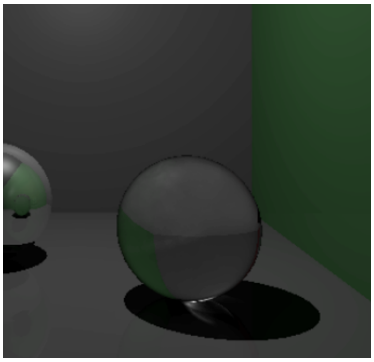


Figure 6: Render with caustics

5.0.4 Indirect soft diffuse

During the rendering process, after raytracing if the object's material is reflective or transmissive, rays are diffusely reflected of the intersecting object. This allows for the visualisation of colour bleeding. This was done by tracing a diffusion ray until intersection, at that point - depending on accuracy preference the value was obtained from the global photon map directly or by sampling.

I attempted to render everything accurately, including indirect diffuse reflections - even though inaccurate was suggested unless it could not be performed. Unfortunately the results of attempting to accurately render the soft diffuse resulted in high amount of noise, as shown in figure 7 and subfigure 8a as a whole scene. To reduce the noise the image was passed through a gaussian low pass filter, the result of this are shown in subfigure 8b. Unfortunately this don't produce the desired effect, I therefore decided to examine the difference between inaccurate and the use of both inaccurate and accurate to render the scene. This is shown in figure 8.

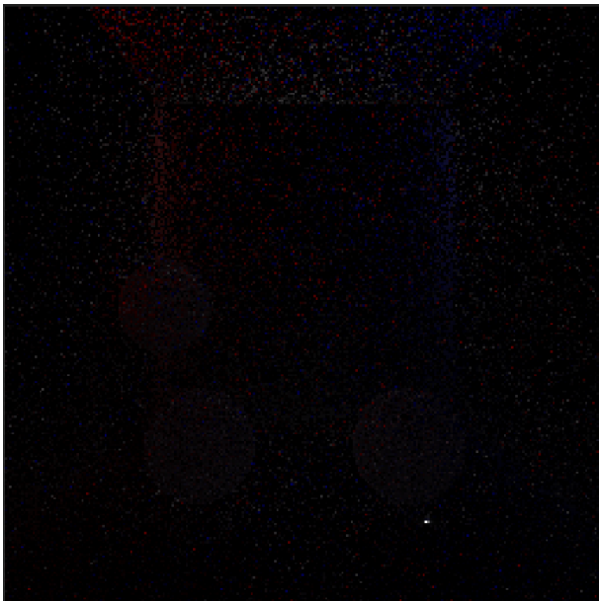
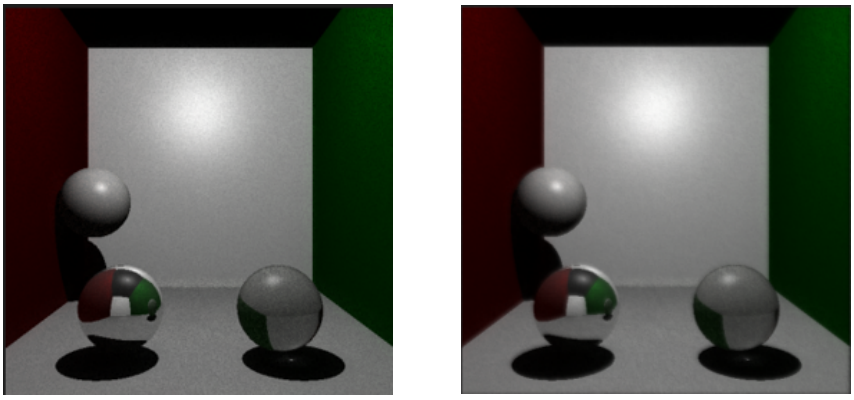


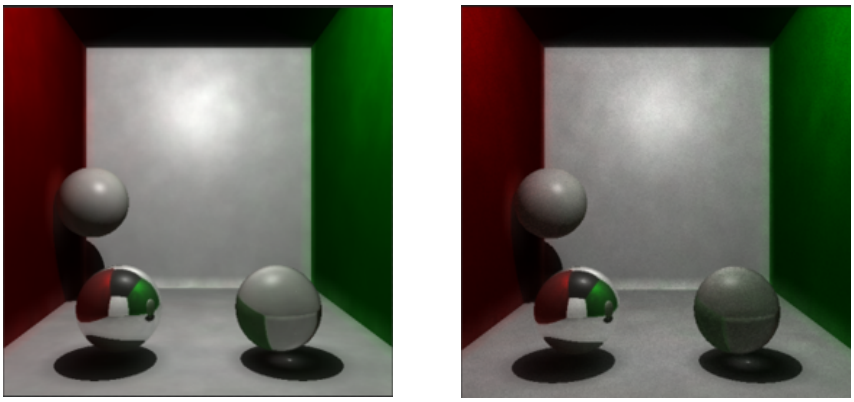
Figure 7: Render showing indirect soft diffuse only

Overall I am undecided which scene I prefer, 9a is very clean, however, 9b seems to have slightly more colour bleeding and the low level noise gives the scene a nice metallic texture. Therefore I decided to use both methods when rendering the scene. To sample a position, the surrounding photon hits were collected and a subset were reversed and their radiance calculated using their originating position , this would only happened to indirect photons otherwise you may sample colours on opposite walls from a central light source unintentionally.



(a) Acurrate soft diffuse (b) Accurate soft diffuse - filtered

Figure 8: Before and after filtering accurate soft diffuse



(a) Inacurrate soft diffuse (b) Combined soft diffuse

Figure 9: Scene comparison using different soft diffuse calculations

List of Figures

References

[1] Y. Yang, "Raytracing," 2019.

[2] K. Cameron, "Photon maps," 2019.

[3] H. W. Jensen, "A practical guide to global illumination using photon mapping," 2002.

[4] A. Project, "Alglib," 2019.