Solving the Rendering equation - Path Tracing

Siddharth Sundar(2015101),
Ramya YS (2015117)

ABSTRACT

Image or scene Rendering is the process of generating an image by means of computer programs. The computer program used to simulate the image contains complex data structures holding geometry, viewpoint, texture, lighting, and shading information as a description of the virtual scene. The data contained in these structures is then passed to a rendering program to be processed and to output the final raster graphics image file.

For photo-realistic rendering, the rendering software utilizes needs to solve the rendering equation, first introduced simultaneously in computer graphics by David Immel et al. and James Kajiya. The following sections outline one solution for the rendering equation, namely, Path Tracing

BACKGROUND

Efficient and realistic Global Illumination is one of the premier problems of Computer Graphics. This is due to the fact that there is no one generic path taken by light rays. Light rays interact with different materials in different ways. A light ray may bounce off of an object, get refracted at the surface, get scattered partially, or be entirely absorbed into the the object. There are infinitely many combinations.

There are two methods possible to trace light rays in a virtual scene,

(i) start from the light source and trace the ray's route til it reaches the eye. This method is called Forward tracing

(ii) trace a ray entering your eye backwards,i.e, shoot a ray from the eye and follow it's path til it reaches the light source it originated from. This method is called Backward tracing

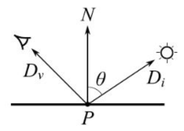
Forward tracing is not efficient, since not all rays originating from the light source enter the eye. Backward tracing is preferred but is not good for indirect lighting.

Simulating indirect lighting is simulating the path taken, by a light ray from the time it is emitted by the light source to the time it enters our eyes . Hence, the algorithms that simulates this path is called light transport algorithms.

OBJECTIVES

- Path tracing for primitives (Spheres, cylinder, triangles)
- Lambertian and Reflective materials
- Loading and rendering meshes
- Textures

METHODS



$$L(P \to D_v) = L_e(P \to D_v) + \int_\Omega F_s(D_v, D_i)|\cos\theta|L(Y_i \to -D_i)\,dD_i$$

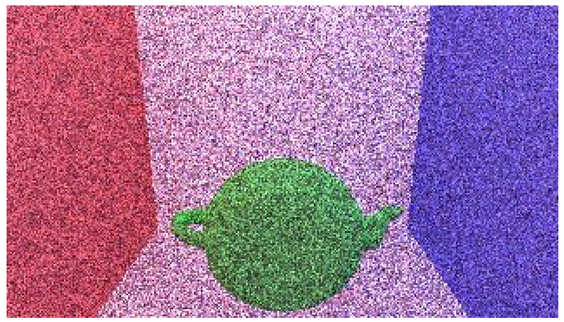
This integral is hard to solve analytically. However, we can obtain a good approximation through the use of Monte Carlo methods. In Monte Carlo methods, we approximate the integral by sampling the integrand at many locations according to a distribution. Then, we add them up weighting them by the inverse of the probability of that location being chosen. More the number of samples, the more closer the sum is to the actual value of the integral. This method is also called unbiased because as the number of samples tends to infinity, we get the actual value of the integral. Biased methods sacrifice this property to achieve better results with fewer samples.

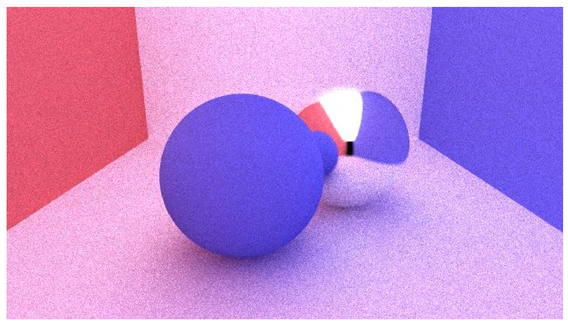
$$\langle L_o(p,\omega_o)\rangle = \frac{1}{N}\sum_{i=1}^{N}\frac{f(p,\omega_o,\omega_i)\,L_i(p,\omega_i)\,\cos\theta_i}{p(\omega_i)}$$

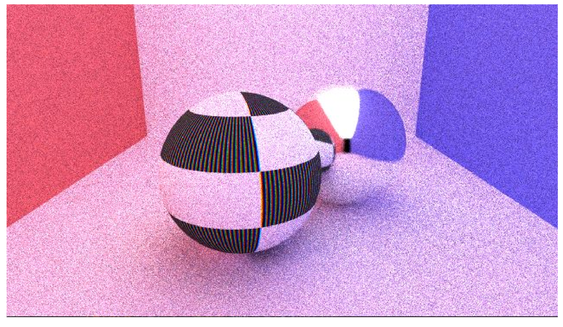
Algorithm 3 Path Tracing Main Loop

```
for each pixel (i,j) do
    Vec3 C = 0
    for (k=0; k < samplesPerPixel; k++) do
        Create random ray in pixel:
            Choose random point on lens P_lens
            Choose random point on image plane P_image
            D = normalize(P_image − P_lens)
            Ray ray = Ray(P_lens, D)
        castRay(ray, isect)
        if the ray hits something then
            C += radiance(ray, isect, 0)
        else
            C += backgroundColor(D)
        end if
    end for
    image(i,j) = C / samplesPerPixel
end for
```

RESULTS







RESULTS

With higher SPP, we get better pictures. The following pictures show mesh rendering, Lambertian/Reflective surfaces and texturing respectively,

CONCLUSIONS

- A major factor with the path tracer is the time is takes to render the scene.
- This is partly due to the reason that the all rays check with each and every object if it intersects it. This becomes especially hard for meshes. Future work would include optimizing this by implementing kd trees to reduce the space where the rays check for intersection. After optimizing, we could try and implement caustics.
- Our path tracer can also be made more parallel. Although using the GPU is a good option, this would require rewriting the recursive ray tracer to an iterative one. A lot of optimizations can be made to make the path tracer faster.