RAY TRACING ENGINE

Mamona Awan, Kwang Hee Ko

Gwangju Institute of Science and Technology,

Gwangju, South Korea.

mona@gist.ac.kr, khko@gist.ac.kr

Abstract: *This paper summarizes the theory and methodology used in making of a simple ray tracing engine. In this project Monte Carlo technique has been used to implement rendering equation to incorporate direct and indirect illumination components of global illumination. The technique uses random samples to solve integration for rendering equation.*

Keywords: *Ray Tracing, Global Illumination, Monte Carlo technique.*

1. Introduction

Global Illumination (GI) is a system that models the path of light as it bounces off a surface onto other surfaces (known as indirect light) rather than just being limited to a surface directlty (known as direct light). Global illumination or indirect illumination is a general term for algorithms used in computer graphics that are meant to add more realistic lighting to 3D scenes. Images rendered using global illumination algorithms are supposed to appear more photorealistic than images rendered using only direct illumination algorithms. However, such images are computationally more expensive as well as consequently much slower to generate. One common approach is to compute the global illumination of a scene by ray tracing [1]. The algorithm stands for casting a ray through each pixel towards the scene and calculating the illumination as it intersects the objects in the scene.

1.1 Ray Tracing

Ray tracing is a technique for generating an image by tracing the path of light through pixels in an image plane and simulating the effects of its intersects with virtual objects. The technique is capable of producing a very high degree of visual realism, usually higher than that of typical scanline rendering methods, but at a greater computational cost. This makes ray tracing suitable for applications where the image can be rendered slowly ahead of time, such as in still images or movies and television visual effects. However, it is unsuitable for real-time applications like video games where speed is critical.

Ray tracing is capable of rendering a vast variety of effects, such as reflection and refraction, scattering, and dispersion phenomena with quite a suitable time cost. The concept of ray tracing is to fire rays from the eye or in our case from a picture, as one ray per pixel, and find the closest object blocking the path of that ray. Using the material properties and the effect of the lights in the scene, this algorithm can determine the shading of this object.

The simplifying assumption is made that if a surface faces a light, the light will reach that surface and not be blocked or in shadow. The shading of the surface is computed using 3D computer graphics shading models for instance phong model. One important advantage ray casting offered over other scanline algorithms is its ability to easily deal with non-planar surfaces and solids, such as cones and spheres. If a mathematical surface can be intersected by a ray, it can be rendered using ray casting. Elaborate objects can be created by using solid modeling techniques and can easily be rendered using this algorithm. Previous algorithms traced rays from the pixels into the scene until they hit an object, but determined the ray color without recursively tracing more rays, hence taking account of only direct illumination.

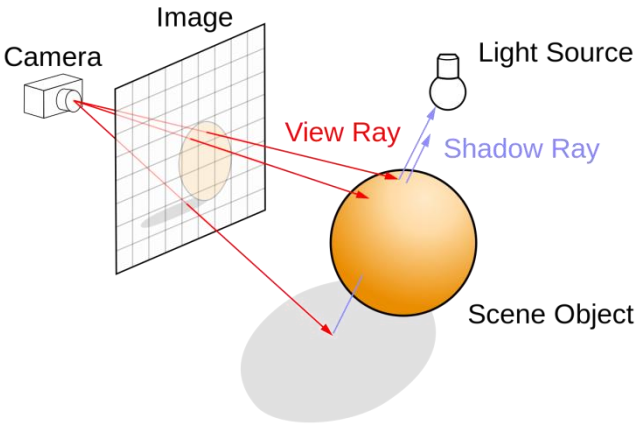

Figure 1: Ray tracing technique for image formation.

The idea is exteneded as when a ray hits a surface, it can generate up to three new types of rays: reflection, refraction, and shadow [2]. A reflection ray is traced in the mirror-reflection direction. The closest object it intersects is what will be seen in the reflection. Refraction rays traveling through transparent material work similarly, with the addition that a refractive ray could be entering or exiting a material. A shadow ray is traced toward each light. If any opaque object is found between the surface and the light, the surface is in shadow and the light does not illuminate it. This recursive ray tracing added more realism to ray traced images.

A major disadvantage of ray tracing is performance. Scan line algorithms and other algorithms use data coherence to share computations between pixels, while ray tracing normally starts the process a new, treating each pixel ray separately [3]. However, this separation offers other advantages, such as the ability to shoot more rays as needed to perform spatial anti-aliasing and improve image quality where needed.

Although it does handle inter-reflection and optical effects such as refraction accurately, traditional ray tracing is also not necessarily photorealistic. True photorealism occurs when the rendering equation is closely approximated or fully implemented. Implementing the rendering equation gives true photorealism, as the equation describes every physical effect of light flow. However, this is usually infeasible given the computing resources required.

The realism of all rendering methods can be evaluated as an approximation to the equation. Ray tracing, if it is limited algorithm, is not necessarily the most realistic. Methods that trace rays, but include additional techniques for instance photon mapping, path tracing, give far more accurate simulation of real-world lighting [4].

It is also possible to approximate the equation using ray casting in a different way than what is traditionally considered to be "ray tracing". For performance, rays can be clustered according to their direction, with rasterization hardware and depth peeling used to efficiently sum the rays.

1.2 Methodology

The implementation in our case has followed the idea of ray tracing. We have considered a picture of specified height and width to be made. As in equation

$$L_{pixel} = \int_0^{imageplane} L(p \rightarrow eye)h(p)dp$$

Where p is pixel on image plane, and h(p) is weighing function. For instance 640x480 pixel resolution is taken into consideration. As for each pixel we start off by creating 4 sub-pixels, for each sub-pixel we take in the N number of samples. Increasing number of sample, results in better estimation for Monte Carlo integration of the estimator of rendering equation. As the rendering equation is

$$L(p \rightarrow \theta) = L_e(p \rightarrow \theta) + L_r(p \rightarrow \theta)$$
$$= L_e(p \rightarrow \theta) + \int_0^{\Omega} L_r(p \leftarrow \varphi) f(p, \theta \leftarrow \varphi) \cos(\varphi, N)\, d\Omega$$

Where Monte Carlo estimator for the integration involved for the calculation of L_r is as following,

$$L_r(x \rightarrow \theta) = \frac{1}{N}\left(\sum_{i=1}^{N} \frac{L(x \leftarrow \varphi) f(x, \theta \leftarrow \varphi) \cos(\varphi, N)}{p(\varphi)}\right.$$

For instance N is taken to be 4. To make the sample random, we take in two random numbers. The random generator takes a random number in range of 0 to 1. This random number is then scaled to be used for random sampling. For each sub pixel we take in 2 random numbers scale them and add to the center of the sub pixel, creating a random location for the ray to be casted. The random direction is also taken by adding random numbers to the direction of the camera and then normalizing it. This ray is then tested if it intersects any of the objects in the scene. If there is no intersection, it returns to generate another ray by the same means and testing until it intersects any object.

As it intersects any object, the distance to that object is stored as how far the ray has traveled to intersect that object. The point where the ray has hit is taken to be the point to cast new rays. These rays are then checked in the same manner if they intersect other objects, if it doesn't intersect any other object and doesn't hit any light source as well, then the pixel is shown as black as if it is in shadow.

For the rays casted from the hit point, if the rays hit the light sources directly, they are given the direct illumination after modulating it with the chance that it will reflect in the direction in which the ray came from. If it hits any other surface the illumination at that point is taken and checked for the part it might add in our illumination.

For the diffuse type of material, the rays scatter over the whole hemisphere over the hit point. Hence the direction of the ray casted from the hit point are taken in consideration by two angles, azimuth in range 0º to 360º and elevation from 0º to 90º. The function calls itself again for the calculation of illumination at that point. The function terminates if the number of reflection increases a certain limit and doesn't hit light source, or the maximum value of the radiance being added gets too small.

For specular reflection, the same rendering equation is used with slight change in selecting the ray direction, as for specular reflection the ray is reflected only in the opposite direction of the arriving ray.

1.3 Pseudo code

The pseudo code for the main loop of the algorithm is as following:

```
For each pixel (i , j)
  Vec3 color = 0;
  For each sub-pixel
    For n number of samples
          Choose random ray casting location
          Choose random ray casting direction
          Cast the ray & check for intersection
          If (intersection ==NULL)
          Return 0;
          Else
            Get intersection point
```

```
            Color += Get illumination at the hit point
            Cast ray from hit point again
            Repeat till converge
          End if
End loop
   Image plane (i,j)=color*1/number of samples;
End loop
```

The pseudo code for illumination at intersected point is as following:

```
For each intersection
 Check reflection type
 If diffuse
   Take random samples over full hemisphere
    Check if it hits light source
    If hits light source
      Add weighted illumination of light source
    Else
     Check if hits object
      If hits object
            Add weighted illumination of hit point
     Else
            Return 0;
If specular
 Take direction of ray in opposite to that of arriving
 Check if it hits light source
    If hits light source
      Add weighted illumination of light source
    Else
     Check if hits object
      If hits object
            Add weighted illumination of hit point
     Else
            Return 0;
```

2. Results

The scene consists of three spheres, among which one is large enough that only a part of it is visible in the picture. Other two spheres are of same radius but have different locations. One is located near the light source, where light source is not visible in the scene, the other one is located to be on the top of the other largest sphere. The colors of the sphere can also be changed.

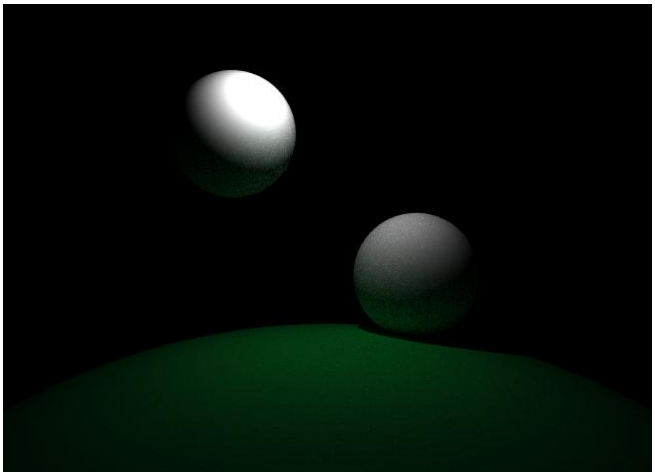

Figure 2: Rendered Scene with diffuse reflection.

In the scene above the direct light and indirect light components can be seen. The green color of the larger sphere has also been accumulated with the indirect light reflected from the diffuse surface to other spheres. The sphere which is closer has more reflected or indirect illumination than the one that is far.

In figure 2, the spheres are given an initial color, now the indirect light illuminates the sphere and cause the original color of the sphere to change a little, the red sphere shows change in color more than the blue sphere. As seen, the blue sphere gets a little illumination on the bottom; however the illumination is still blue.

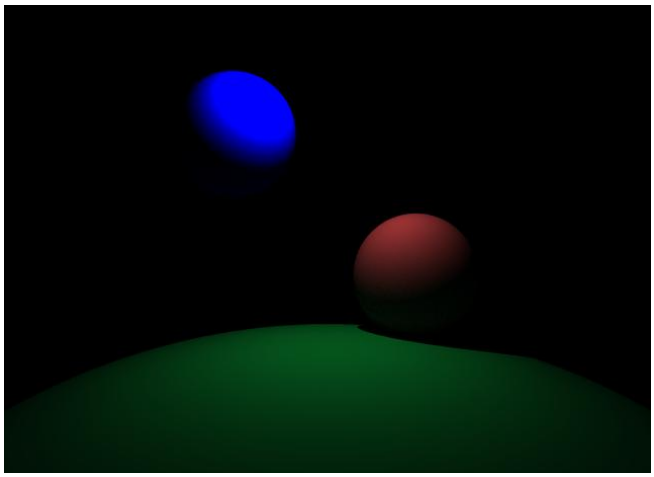

Figure 3: Rendered Scene with diffuse reflection and different color of spheres.

Introducing specular reflection in the same instance as in figure 1 renders as shown in figure 3. As seen, the green sphere reflects on the surface of specular reflecting sphere. The reflection of other sphere can also be seen, the reflection visibly shows the illuminated as well as the greener part of the sphere.

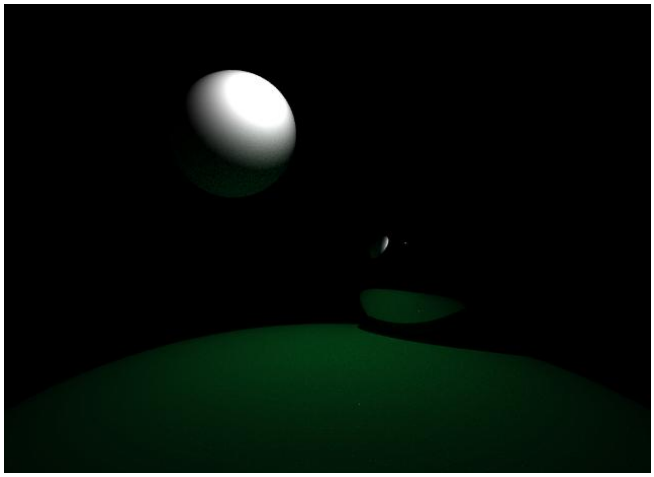

Figure 4: Rendered Scene with specular reflection on one sphere and diffuse on others.

Increase in number of samples increases the quality of rendered image, as it decreases the noise. The results for different sample numbers chosen are shown below.

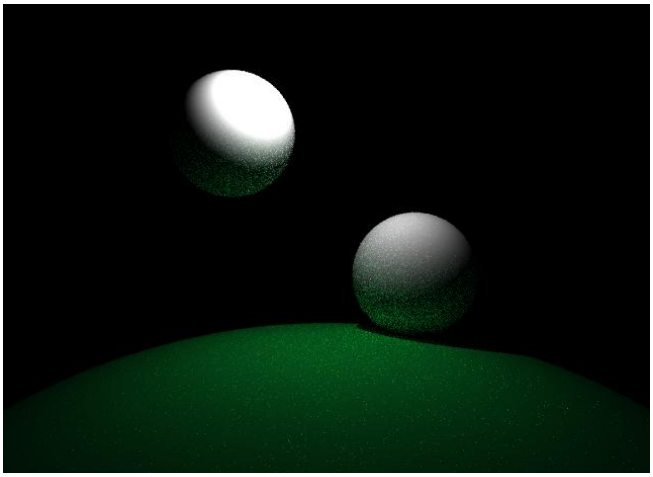

Figure 5: Rendered Scene with 1 sample per sub-pixel.

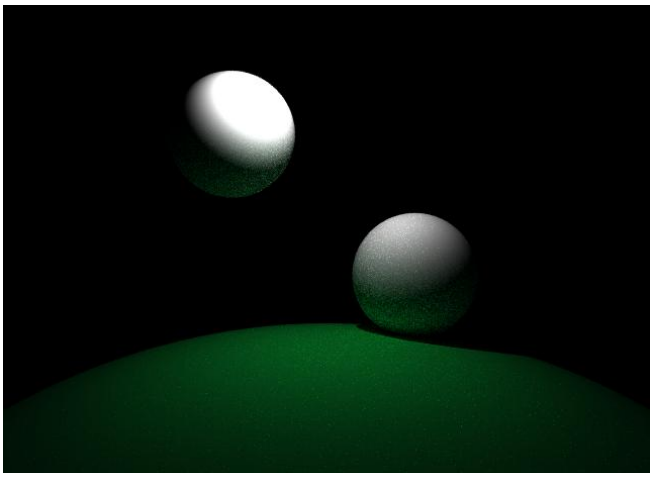

Figure 6: Rendered Scene with 10 samples per sub-pixel.

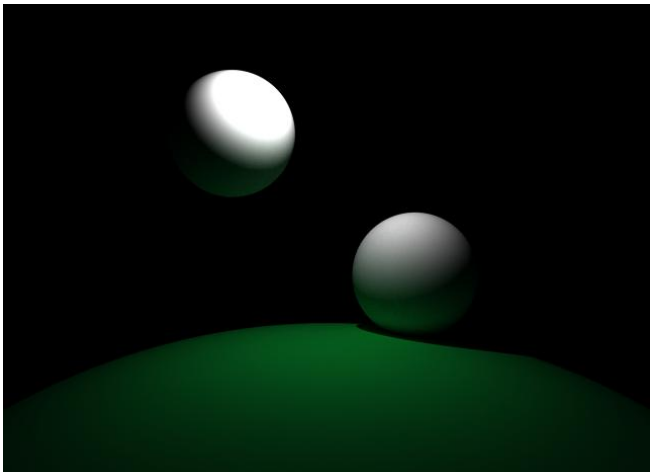

Figure 6: Rendered Scene with 100 samples per sub-pixel.

The difference in noise is evident from the images above. As the number of samples is increased the surfaces of the objects appear smoother and the illumination appears to be smoothly changing.

6. Conclusions

The program is a small implementation of the ray tracing algorithm. The results show that by increasing the number of samples, fairly good quality images can be rendered by this algorithm, while taking relatively more rendering time. This makes it unsuitable for real-time applications. However this algorithm can be efficiently used to render images for non- real time applications.

References

[1] Andrew S. Glassner, *An overview of ray tracing*, 1989

[2] Arichunan, Navaladi, et al, *Ray Tracing in Computer Graphics*, 2012

[3] Van Dam, Andries, et al, *Introduction to computer graphics. Vol. 55. Reading: Addison-Wesley*, 1994