CG_7_Path Tracing Report

PB21010479 Caoliwen Wang

April 21, 2024

1 Problem Statement

In our last task, we implemented a model for local lighting and the rendering of shadows. However, such a lighting model is evidently very limited. The lighting at a single point is highly complex, and the influence of light from different types and directions on colors is also extremely intricate. Therefore, in this task, we need to implement a more sophisticated rendering model, which will require numerous complex processing techniques.

2 Propose Algorithms

2.1 Physics Based Rendering

To produce images with lighting and shadows that closely resemble the real world, we need to consider how light interacts with objects in the physical world and then enters the human eye, creating the colorful and beautiful world we perceive. Due to space constraints, we will skip the introduction of geometric optics and wave optics knowledge, and only introduce some physical concepts in light rendering techniques.

2.1.1 Irradiance

The irradiance $L(x, \omega)$ of a light ray is defined as the power per unit area, where x is the spatial position and ω is the direction along which it is measured. The irradiance can be denoted symbolically.

2.1.2 BRDF Function

In the real world, objects of different materials exhibit a wide variety of reflection patterns when interacting with light. To describe how materials reflect incident light at a given spatial position, we use the Bidirectional Reflectance Distribution Function (BRDF) $f_r(x, \omega_i, \omega_o)$.

2.1.3 Rendering Equation

The rendering equation describes the radiance at a point on a surface in a scene. It states that the outgoing radiance at a point in a given viewing direction is equal to the sum of emitted radiance at that point and the incoming radiance from all directions adjusted by the BRDF

function. This equation is crucial for rendering images as it describes how light propagates and interacts within a scene.

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega f_r(x, \omega_i, \omega_o) \cdot L_i(x, \omega_i) \cdot (\omega_i \cdot n) \, d\omega_i \tag{2.1}$$

- $L_o(x, \omega_o)$ is the outgoing radiance from point x in the direction ω_o,
- $L_e(x, \omega_o)$ is the emitted radiance at point x in the direction ω_o,
- $f_r(x, \omega_i, \omega_o)$ is the Bidirectional Reflectance Distribution Function (BRDF) at point x for incoming direction ω_i and outgoing direction ω_o,
- $L_i(x, \omega_i)$ is the incoming radiance at point x from direction ω_i,
- $(\omega_i \cdot n)$ is the cosine factor, where n is the surface normal,
- Ω represents the hemisphere of incoming directions.

Interacting light rays with the scene is crucial, as illustrated in Figure 1.

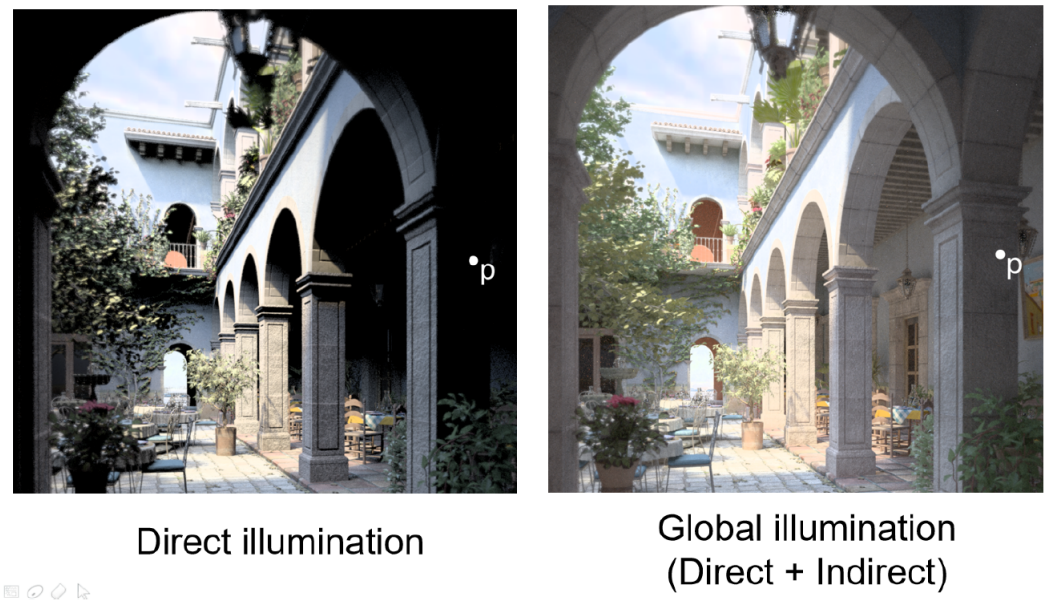

Figure 1: Interacting Light Rays with Scene

In the case of global illumination alone, as depicted, the scene appears dim, with many details of objects not directly illuminated by the light source hidden in darkness. However, with the combination of global illumination and local illumination, the overall scene becomes brighter, and it can illuminate details of objects that were not visible in the left image.

2.2 Ray Tracing

The ray tracing algorithm is an important early rendering method in computer graphics. As the name suggests, its core idea is to trace rays of light from the camera to find light sources. Specifically, a ray of light is cast from each pixel on the camera plane. This ray undergoes a series of reflections/refractions in the scene before eventually reaching either a light source or the background. Light that reaches the background at the pixel presents the color of the corresponding position on the background (multiplied by a series of factors along the path), while light reaching the light source presents the color of the light source at that point (also multiplied by a series of factors along the path). The core principle is illustrated in Figure 2.

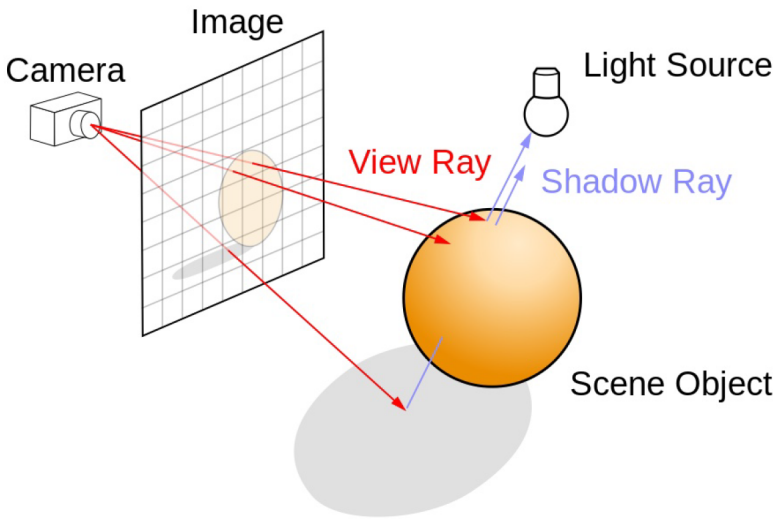

Figure 2: Ray Tracing

The main steps of the original algorithm (Whitted-style) are as follows:

```
color trace(point p, vector d, int step)
{
    color local, reflected, transmitted;
    point q;
    normal n;
    if(step > max) return(background_color);
    q = intersect(p,d,status);
    if(status == light_source)
    return(light_source_color);
    if(status == no_intersection)
    return(background_color);
    n = normal(q);
    r = reflect(q,n);
    t = transmit(q,n);
    local = phong(q,n,r);
    reflected = trace(q,r,step+1);
    transmitted = trace(q,t,step+1);
```

```
18         return(local+reflected+transmitted);
19     }
```

The main shortcomings of the ray tracing algorithm are as follows:

1. When the light source area is small, the probability of rays emitted from the camera reaching the light source decreases, leading to many rays being wasted as they cannot reach the light source, resulting in a large number of noise points. As shown in the Figure 3.

2. The ray tracing algorithm actually does not solve the rendering equation; instead, it is based on simple principles of ray reflection/refraction. Therefore, the physical accuracy of the results cannot be guaranteed.

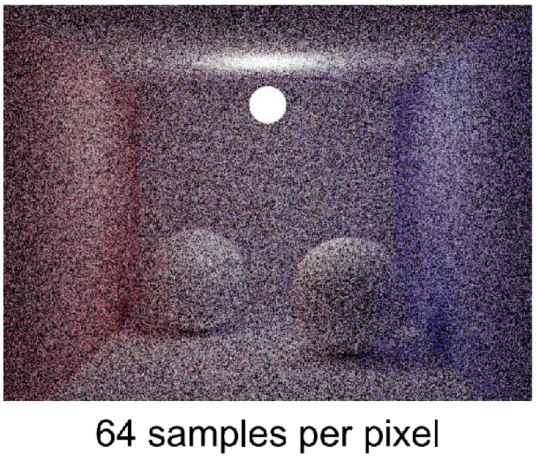

Figure 3: Ray Tracing Problem

2.3 Monte Carlo sampling method

The Monte Carlo sampling method is a numerical computation method based on probability statistics, commonly used to simulate random phenomena. In graphics, Monte Carlo methods are often used to estimate complex lighting effects in ray tracing, such as global illumination and indirect lighting.

This approach approximates solutions by generating a large number of random samples, thereby compensating for the shortcomings of traditional ray tracing algorithms to some extent. Monte Carlo sampling can effectively reduce noise and improve the quality and realism of rendering results.

The description of the problem is as follows:

We need to solve the numerical integral as follows:

$$I = \int_a^b f(x)\, dx.$$

We can estimate the value of this integral using a computer through sampling methods. Our goal is to construct random variables to obtain an unbiased estimate, denoted as

$$\hat{I} = \frac{f(x)}{p(x)}.$$

In fact,

$$E[\hat{I}] = E[\frac{f(x)}{p(x)}] = \int_a^b \frac{f(x)}{p(x)} p(x)\, dx = \int_a^b f(x)\, dx$$

This way, we obtain an unbiased estimate of $\hat{I}$.

However, in practice, we often do not know the probability distribution function, only can estimate it. The most commonly used estimations include the uniform distribution. Suppose we sample n samples, $x_1, x_2, \ldots, x_n$,

$$\hat{I}_j = \frac{f(x_j)}{p(x_j)}.$$

Then we obtain an estimate of the original integral as follows:

$$\bar{I} = \frac{1}{n} \sum_{j=1}^{n} \hat{I}_j.$$

This estimate satisfies:

$$\int_a^b f(x)\, dx = \lim_{n\to\infty} \bar{I} = \lim_{n\to\infty} \frac{1}{n} \sum_{j=1}^{n} \hat{I}_j.$$

We can also introduce parameters p_i according to the importance of different sampling points, satisfying:

$$\sum_{j=1}^{n} \omega_j = 1.$$

Finally:

$$\bar{I} = \sum_{j=1}^{n} \omega_j \frac{f(x_j)}{p(x_j)}.$$

This is the core idea of using Monte Carlo sampling algorithm to solve numerical integration.

2.4 Pace Tracing

Path tracing algorithm is a ray tracing method based on the Monte Carlo sampling algorithm. Its core idea is similar to ray tracing, which is to trace rays from the camera to find light sources. However, in path tracing, multiple rays are emitted from each pixel on the camera plane to search for light sources. When rays intersect with objects in the scene and undergo reflection, a direction is sampled from the hemisphere according to a predetermined probability distribution function (referred to as BRDF sampling), and this probability factor is then removed from the final expression. The rest of the process is consistent with ray tracing.

The essence of the path tracing algorithm is to combine the contributions of multiple light paths to the lighting effect according to their probabilities, thus realizing the rendering equation. The core algorithm is shown in Figure 4.

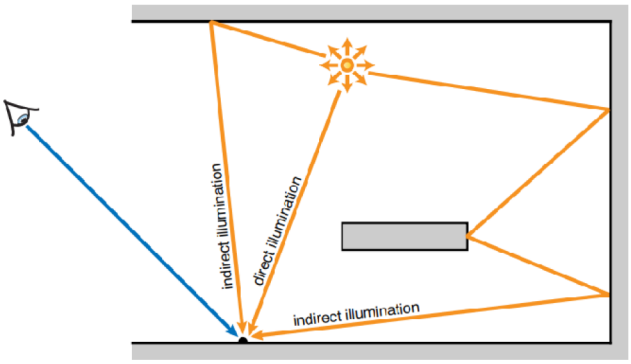

Figure 4: Path Tracing

It's worth mentioning that since this is a recursive algorithm, it must terminate. To ensure termination of the iterative reflection of rays, we can introduce the technique of Russian Roulette: generating a random number at each iteration to determine whether to proceed with the next reflection of the ray, thus preventing the possibility of infinite iteration.

The algorithm proceeds as follows:

```
shade(p, wo)
    Test Russian Roulette with p_rr, if fail then return 0.0;
    Randomly choose one direction wi with pdf(wi);
    Trace a ray r(p, wi);
    if ray r hit the light:
        return L_i * f_r * cos / pdf(wi) / p_rr;
    else if ray r hit an obj at position q:
        return shade(q, -wi) * f_r * cos / pdf(wi) / p_rr;
ray_generation(camPos, pixel)
    Uniformly choose N sample postitions within pixel;
    pixel_radience = 0.0;
    for each sample in the pixel:
        Shoot a ray r(camPos, cam_to_sample_direction);
        If ray r hit the scene at p:
        pixel_radience += 1/N * shade(p, cam_to_sample_direction);
    return pixel_radience;
```

However, it is worth noting that the path tracing algorithm does not solve the problem when the light source area is very small, and emitted rays cannot find the light source. A natural approach to address this issue is: since rays emitted from the camera cannot reach the light source, can we directly emit rays from the light source to reach objects and directly calculate global illumination? Following this idea, we can improve the efficiency of the path tracing algorithm by sampling the light source. As shown in the following Figure 5.

At this point, by geometric relationships, the integral over the BRDF hemisphere is trans-

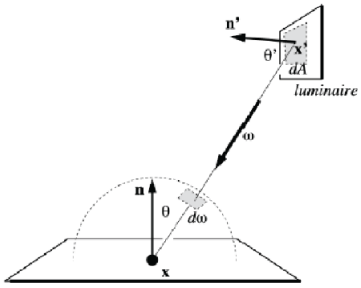

Figure 5: Path Tracing Problem and Solution

formed to the plane A where the area light source is located:

$$L_0(x,\omega_0) = \int_\Omega L_i(x,\omega_i) f_r(x,\omega_i,\omega_o)\cos\theta \mathrm{d}\omega_i = \int_A L_i(x,\omega_i) f_r(x,\omega_i,\omega_o)\frac{\cos\theta\cos\theta'}{||x'-x||^2}\mathrm{d}A. \quad (2.2)$$

In the improved path tracing algorithm, the main process of shading is as follows:

```
shade(p, wo)
    // direct light, sampling the light
    Uniformly sample the light at x with pdf_light = 1/A;
    Check visibility, if invisible L_dir = 0;
    else L_dir = L_i * f_r * cos(theta1) *
cos(theta2) / |x - p|^2 / pdf_light;

    // indirect light, BRDF sampling
    L_indir = 0.0;
    Test Russian Roulette with p_rr
    Randomly choose one direction wi with pdf(wi);
    Trace a ray r(p, wi);
    if ray r hit an non-emitting obj at position q:
        return shade(q, -wi) * f_r * cos / pdf(wi) / p_rr;
    return L_dir + L_indir;
```

3 Programming

3.1 Node programming

In this assignment,The node used in this instance is very simple. We adopted the connection method as shown in Figure 6.

4 Experimental Results

I think the results of this algorithm are fantastic.

We perform testing using the cornell scene. The result is as shown in the Figure 7.

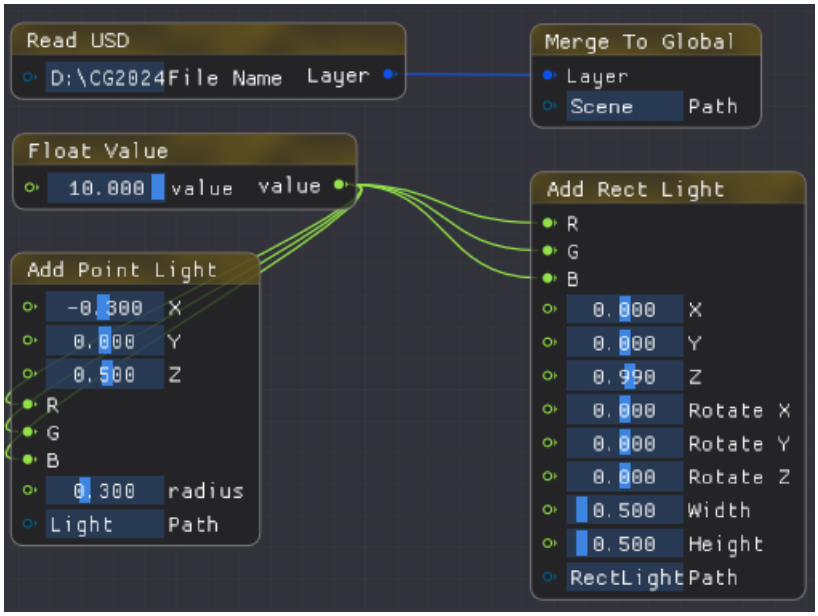

Figure 6: Render Nodes

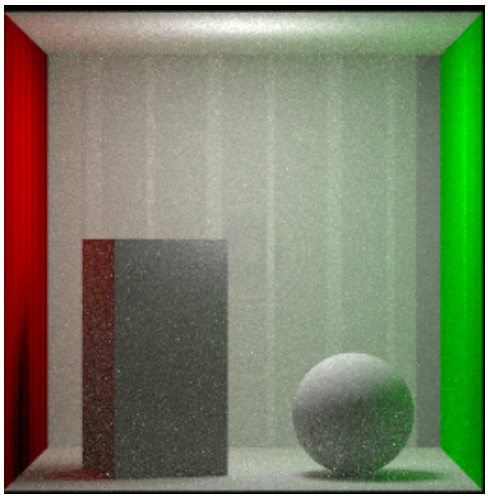

Figure 7: Cornell with Rec and Point Light

We can observe that there are indeed some noise artifacts present. Issues related to noise can be adjusted by modifying the spp (samples per pixel) value in the integrator file; a higher spp value leads to fewer noise artifacts. We can also notice a slight red tint on the left side of the cuboid and a slight green tint on the right side of the sphere, which is exactly what we want.

If we use the first renderer, we would achieve the result shown in Figure 8, which is obviously less realistic in comparison.

For the cases of using only area light sources and using only point light sources, our comparative test results are shown in Figures 9 and 10, respectively.

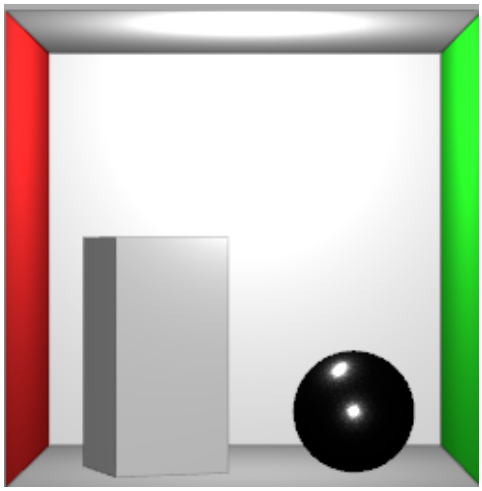

Figure 8: Cornell Using another renderer

5 Summary

This experiment involved more complex mathematical knowledge and did indeed yield good results. However, I noticed that even when obtaining the displayed experimental results, there were still cases where extremely poor pixel values appeared. I believe this is related to the probabilistic nature of the selection process. If one iteration produces poor results, it may be necessary to run several more iterations, which is crucial.