Ray-Tracing Project Assignment Report

Peizhi(Paige) Li

July 31, 2023

1 Abstract

This report presents the implementation of a ray-tracer, showcasing fundamental ray-tracing techniques such as the basic ray-tracing algorithm, Phong shading, shadow rays, and recursive reflection. The github link to the project code can be found at github.com/lipz0624/cis_raytracer

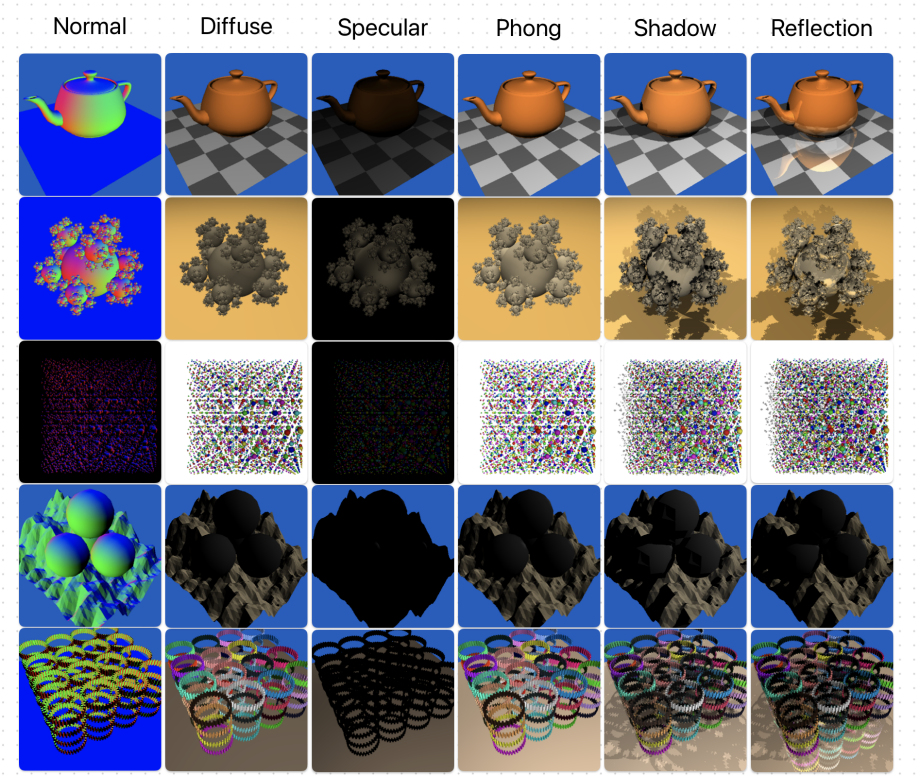

Figure 1: The Results

2 Key Steps and Code Implementation

2.1 Ray-Tracing Overview

Ray tracing is a rendering technique in computer graphics that simulates the behavior of light by tracing rays from a virtual camera into a 3D scene. It calculates how these rays interact with objects, surfaces, and light sources, producing highly realistic images with advanced lighting effects, shadows, reflections, and refractions. By casting primary rays from the camera, finding intersections with objects, and recursively tracing secondary rays for shadows and reflections, ray tracing achieves photorealistic results, making it a widely used method in video games, movies, and architectural visualization for generating visually stunning and lifelike images.

2.2 Ray Intersection

The primary rays are tested for intersection with the objects in the scene, such as triangles for 3D models or spheres. This involves finding the point(s) where the ray intersects with the surface of an object.

```
SlVector3 Tracer::trace(const Ray &r, double t0, double t1) const {
    HitRecord hr;
    hr.bHit = false;
    SlVector3 color(bcolor);

    bool hit = false;

    // Step 1 See what a ray hits
    for (int i = 0; i < surfaces.size(); i++){
        HitRecord hrTemp;      You, 2 days ago • Initial push
        hrTemp.bHit = false;
        hit = surfaces[i].first->intersect(r, t0, t1, hrTemp);
        hrTemp.f = surfaces[i].second;
        // compare hr.t
        if (hrTemp.bHit) {
            if (hrTemp.t < t1) {
                hr = hrTemp;
                t1 = hrTemp.t;
            }
        }
    }

    if (hr.bHit) {
        // set HitRecord
        hr.p = r.e + hr.t * r.d;
        hr.v = r.e - hr.p;
        normalize(hr.v);
        hr.raydepth = maxraydepth;
        //color = hr.n;
        color = shade(hr);
        color += hr.f.ks * reflect(r, hr);
    }
    return color;
}
```

Figure 2: The Code Implementation of Trace

2.2.1 Triangle Intersection

This function tests for ray-triangle intersection. It takes a Ray object r, representing a ray with an origin and direction, and two doubles $t0$ and $t1$ representing the valid range for the intersection distance. The function calculates whether the ray intersects with the triangle defined by its vertices a, b, and c. If an intersection is found within the valid range, it updates the HitRecord object hr with information about the intersection point, normal, and other properties related to shading. TrianglePacth calls the base class intersect function from Triangle and then computes the interpolated normal at the intersection point using barycentric coordinates $\alpha, \beta, and\ \gamma$.

$$\mathbf{V}_1 + \beta(\mathbf{V}_2 - \mathbf{V}_1) + \gamma(\mathbf{V}_3 - \mathbf{V}_1) = \mathbf{A} + t\,\mathbf{D}$$

Figure 3: The Ray-Triangle Intersection

$$\begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix}.$$

$$x = \frac{\begin{vmatrix} d_1 & b_1 & c_1 \\ d_2 & b_2 & c_2 \\ d_3 & b_3 & c_3 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix}}, \quad y = \frac{\begin{vmatrix} a_1 & d_1 & c_1 \\ a_2 & d_2 & c_2 \\ a_3 & d_3 & c_3 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix}}, \text{ and } z = \frac{\begin{vmatrix} a_1 & b_1 & d_1 \\ a_2 & b_2 & d_2 \\ a_3 & b_3 & d_3 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix}}.$$

Figure 4: The Cramer's Rule

2.2.2 Sphere Intersection

This function tests for ray-sphere intersection. It takes a Ray object r, representing a ray with an origin and direction, and two doubles t0 and t1 representing the valid range for the intersection distance. The function calculates whether the ray intersects with the sphere defined by its center c and radius rad. If an intersection is found within the valid range, it updates the HitRecord object hr with information about the intersection point, normal, and other properties related to shading.

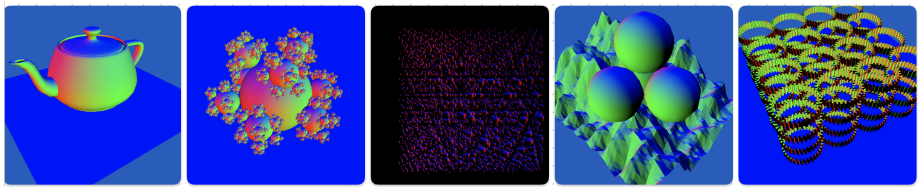

Figure 5: The Nomal Maps of Given Input Files

2.3 Shading and Lighting

Once an intersection is found, the renderer calculates the shading of the surface at that point. This includes determining the color of the surface, applying lighting models (e.g., Phong reflection model) to simulate how light interacts with the material, and calculating surface normals for reflections and refractions.

$$R = k_a I + k_d I \max(\hat{\mathbf{l}} \cdot \hat{\mathbf{n}}, 0) + k_s I \max(\hat{\mathbf{r}} \cdot \hat{\mathbf{v}}, 0)^p$$

Figure 6: The Phong Shading

where: K_a is the ambient coefficient of the setting. I is the color and intensity of the light source. K_d is the diffuse coefficient of the material. n is the normalized surface normal. l is the normalized incident light vector. $(l \cdot n)$ represents the dot product between the surface normal and the incident light vector. K_s is the specular coefficient of the material. r is the normalized reflected light vector. v is the normalized view vector. p is the shininess exponent, controlling the size and intensity of the specular highlight. $(r \cdot v)$ represents the dot product between the reflected light vector and the view vector.

2.4 Shadow Rays

Shadow rays cast from an intersection point towards a light source to determine if the point is in shadow or illuminated by that light. When a primary ray intersects with an object in the scene, the ray tracer needs to determine whether the point of intersection is in direct line of sight with a light source or if it is occluded by another object. For each point of intersection on a surface that receives light, shadow rays are cast towards all light sources in the scene. If any of these shadow rays intersect with an object before reaching the light source, it means the point is in shadow, and no direct illumination from that light source is received at that point. On the other hand, if none of the shadow rays intersect with any objects, the point is directly illuminated by the light source, and the surface can be illuminated accordingly using shading models like the Phong reflection model.

2.5 Reflection Rays

Reflection rays cast from an intersection point to simulate the reflection of light off reflective surfaces. When a primary ray intersects with a reflective surface (e.g., a mirror or a glossy material), a reflection ray is traced to simulate how light bounces off the surface in a predictable manner. Reflection rays are calculated by determining the direction of reflection based on the surface normal at the intersection point and the direction of the incident ray. The direction of reflection follows the "angle of incidence equals angle of reflection" principle, which is governed by the law of reflection. After tracing the reflection

ray, the ray tracer recursively continues the ray tracing process from the new intersection point, again calculating the shading and determining whether the ray should generate more reflection rays. The recursion can continue up to a specified maximum depth or until the reflected ray intensity becomes negligible. Recursive reflection allows the ray tracer to simulate multiple levels of reflections, creating visually appealing and realistic rendering effects, especially in scenes with highly reflective materials or mirrors.

```
function reflect(ray, hr):
if hr.raydepth == 0:
    # Return black color if recursion depth is 0
    return SlVector3(0.0)

newDir = normalize(ray.d - 2 * dot(ray.d, hr.n) * hr.n)
reflectedRay = Ray(hr.p, newDir)

closestHR = null
closestT = INFINITY

# Intersect with each object to find the closest intersection
for each surface in surfaces:
    hitRecord = intersect(surface, reflectedRay, shadowbias, closestT)
    if hitRecord.bHit:
        closestHR = hitRecord

if closestHR is not null:
    # Calculate the reflection color recursively with reduced depth
    closestHR.raydepth = hr.raydepth - 1
    refCor = reflect(reflectedRay, closestHR)

    return shade(closestHR) + closestHR.f.ks * refCor

return SlVector3(0.0)
```

2.6 Optimization

Anti-aliasing and jittering are used in this project to reduce visual artifacts and improve the overall image quality.

Anti-aliasing is a process used to reduce the "aliasing" effect, also known as the "jagged edges" or "staircase effect." Anti-aliasing aims to smooth out these jagged edges and create a more visually pleasing and realistic image. Instead of treating each pixel as a single point, multiple samples are taken within each pixel, and the color values are averaged to obtain a smoother, more accurate

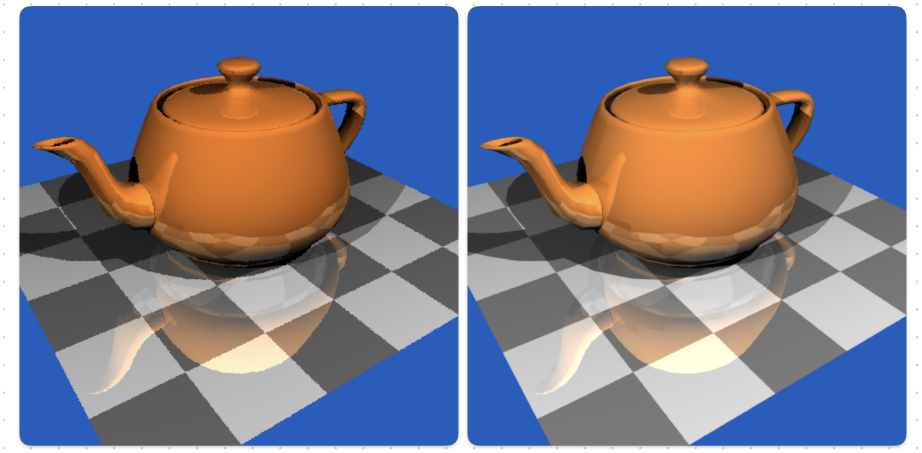

Figure 7: The Normal vs. Anti-aliasing Teapot.

representation of the scene. The number of samples per pixel is determined by the anti-aliasing technique used (e.g., used 8x sampling in my project).

Jittering is a technique used in conjunction with anti-aliasing to further improve the quality of the rendered image. It helps reduce regular patterns that can be introduced by the anti-aliasing process, such as grid-like artifacts or "moire" patterns. In jittering, the locations of the samples within each pixel are randomly displaced to distribute them more evenly across the pixel area. By introducing random offsets to the sample positions, jittering helps break up the regular patterns that could arise from fixed sampling patterns.

3 Discussion

Ray tracing produces highly realistic images with accurate lighting and shading effects. It can simulate complex interactions such as reflections, refractions, and shadows, resulting in visually stunning and lifelike scenes. However, it is computationally demanding, especially for scenes with numerous light sources, complex geometry, and extensive reflection/refraction recursion. Real-time ray tracing is challenging to achieve without specialized hardware. Rendering high-quality ray-traced images can take significant time, especially for animations or large-scale scenes. This can limit its use in time-critical applications.

To address the computational complexity of ray tracing, we can explore and implement various optimization techniques. Techniques like bounding volume hierarchies (BVH), spatial data structures, and adaptive sampling can significantly reduce rendering times. Additionally, we can utilize parallel processing and leverage GPU or specialized hardware support for ray tracing to improve

rendering performance and enable real-time or interactive ray tracing experiences.