# TNCG15 Renderer

Oliver Lundin[1]

**Abstract**
This report explores the implementation of an advanced ray-tracing renderer with global illumination capabilities to achieve photorealistic image rendering. The project applies Monte Carlo integration and ray-tracing techniques to calculate complex light interactions within a 3D scene, including reflections, refractions, and diffuse color bleeding. The model, developed in C++, is designed to handle multiple light bounces, allowing for accurate simulation of indirect lighting. Results demonstrate the renderer's effectiveness in creating realistic images, though at a high computational cost.

**Source code**: https://github.com/olvard/raytracing

**Authors**
[1] *Media Technology Student at Linköping University, olilu316@student.liu.se*

## Contents



## 1. Introduction

The developmnt of ray tracing and global illumination techniques has a rich history, beginning with implementations by Turner Whitted, who introduced a ray tracing algorithm in his paper [1]. While Whitted's model enabled reflections, refractions, and shadows, it simulated only direct lighting, meaning there was no simulation of light bouncing between surfaces. Years after the initial paper, advancements were made, including using monte-carlo integration for tracing all the ray paths. This enabled global illumination.

This project aims to explore the possibilites of achieving photo-realistic images using a global illumination model based on monte-carlo raytracing. Ray tracing provides the foundational method for calculating light paths, while global illumination extends this to simulate realistic light interactions throughout a scene. By integrating both techniques, modern rendering systems achieve a high level of realism.

## 2. Theory

This section explores the theory needed for implementing a renderer.

### 2.1 Ray-tracing

The fundamental concept of ray-tracing is shooting a ray into a scene, from a light source, or an "eye" (the viewing point of the scene). When a ray intersect with an object it evaluates the light contribution from any direct light sources in the scene. To determine if the point is in direct light, shadow rays are cast from the intersection point to each light source. If a shadow ray reaches a light source without being blocked, the point is illuminated by that light source. Otherwise, it falls into shadow.

An important part of ray-tracing is the recursive nature of it. When a ray hits an object there will be new rays spawning from that hit point depending on the objects surface.

As the ray intersects with different objects it adds up "radiance". Radiance is representing the amount of light passing through or emitted from a specific point in a given direction per unit area per unit solid angle. Each bounce or interaction contributes to the final radiance value that will color the

pixel. By integrating or summing the contributions from direct lighting, indirect lighting, reflections, and refractions along the ray's path, a final radiance value for each pixel can be calculated.

### 2.2 Global Illumination

The problem with early rendering models is that they do not take their surrounding into account. For example whitted used the Phong shading model[1], which is considered a local lightning model since it does not simulate the bounces between surfaces. It calculates diffuse, specular, and ambient components of light on each surface hit by a ray. In 1980 this was the most computationally efficient way of achieving realistic renders. A more recent model is called the "rendering equation"[2].

$$L(x \to \theta_o) = L_e(x \to \theta_o) + \int_\Omega f(x, \theta_i, \theta_o)\, L(x \leftarrow \theta_i) \cos\Omega_i \, d\theta_i \tag{1}$$

This equation includes:

- $L(x \to \theta_o)$: The outgoing radiance at point $x$ in direction $\theta_o$.
- $L_e(x \to \theta_o)$: The emitted radiance from point $x$ in direction $\theta_o$.
- $f(x, \theta_i, \theta_o)$: The bidirectional reflectance distribution function (BRDF).
- $L(x \leftarrow \theta_i)$: The incoming radiance at $x$ from direction $\theta_i$.
- $\cos\Omega_i$: The cosine term accounting for the angle between the incoming direction $\theta_i$ and the surface normal.

The rendering equation calculates the total light at a point by combining direct light (light coming directly from a source) and indirect light (light that has bounced off other surfaces). The BRDF models the behavior of light as it interacts with different materials, accounting for varying reflectivity and transparency,. The rendering equation also accounts for scattering within translucent materials and absorption within partially transparent materials, allowing for complex light behaviors like caustics in glass.

To solve the rendering equation, Monte Carlo integration is used to approximate it by tracing thousands or millions of random light paths.

### 2.3 Intersection algorithms

To determine if a ray has intersected with an object, different intersection algorithms can be used. For rectangle polygons the ray-plane intersection equation can be solved by projecting the ray onto the plane using the dot product. The dot product between the plane's normal vector and the ray direction gives us information about the angle between them. If this dot product is negative, it indicates that the ray is approaching the plane and can intersect it.

$$\vec{N} \cdot \vec{D} = \cos(\theta) \tag{2}$$

$$t = \frac{\vec{N} \cdot (\vec{P_0} - \vec{O})}{\vec{N} \cdot \vec{D}} \tag{3}$$

Once we compute the parameter t (the distance from the ray's origin), we substitute it into the ray's parametric equation to get the exact point on the plane where the intersection occurs.

$$\vec{P_{intersection}} = \vec{O} + t\vec{D} \tag{4}$$

Even if the ray intersects the plane, we still need to verify that the point of intersection is inside the bounds of the rectangle. This is done by checking whether the coordinates of the intersection point lie within the edges of the rectangle.

For triangles the Möller-Trumbore algorithm can be used. It uses barycentric coordinates to express the point of intersection in relation to the triangle's vertices. Barycentric coordinates are a way of representing a point within a triangle as a weighted sum of the triangle's three vertices. The weights tell us how far the point is from each vertex.

The algorithm computes the point of intersection by solving a system of linear equations that relate the ray direction and the edges of the triangle. Once an intersection is found, the barycentric coordinates are used to check if the point lies inside the triangle. If the coordinates are all positive and their sum is 1, the point is within the triangle.

### 2.4 Surfaces

The type of surface will determine what happens to the ray next. The three main surfaces are diffuse, perfect reflective surfaces or refractive surfaces. A diffuse surface will generate a new ray with a new random direction in the local hemisphere [2]. A reflective surface will create a new ray with a new direction determined according to the law of reflection and the direction of the outgoing ray is given by equation 5

$$\vec{d} = \vec{d} + 2(\vec{p} - \vec{x}) = \vec{d} - 2(\vec{d} \cdot \vec{N})\vec{N} \tag{5}$$

For refractive surfaces, the direction of the refracted and reflected ray needs to be computed [2]. Let $d_o$ be the unit vector from the eye to $x$. The normal is N. Taking a unit vector $d_o$ is needed for the refracted ray. The direction of the incoming reflected ray is

$$\vec{d}_{\text{refl}} = \vec{d_o} - 2(\vec{d_o} \cdot \vec{N})\vec{N} \tag{6}$$

The angle of the refracted ray can be computed using snell's law.

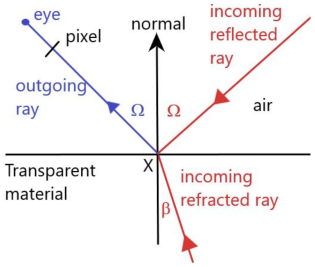


**Figure 1.** Visual of how refraction is handled.

$$\sin\beta = \frac{\sin\Omega}{n_2} = R^{-1}\frac{\sin\beta}{n_1} n_1 = 1.0, \quad n_2 = 1.5 \qquad R = \frac{1}{1.5}. \tag{7}$$

The direction of the refracted ray then becomes:

$$\vec{d} = R\vec{d} + \vec{N}\left(-R(\vec{N}\cdot\vec{d}) - \sqrt{1 - R^2\left(1 - (\vec{N}\cdot\vec{d})^2\right)}\right) \tag{8}$$

## 3. Method

To further investigate the possibilities of a global illumination ray-tracing model a custom one was implemented. The model that was implemented relies heavily on the concepts and theories described in the previous chapter and was built in C++. And runs on the CPU.

### 3.1 Scene and Objects

To test the capabilities of the model a scene was configured. The scene consists of different types of objects that can represent the most interesting surfaces to be showcased for the model. A simple room with hexagonal form was constructed.

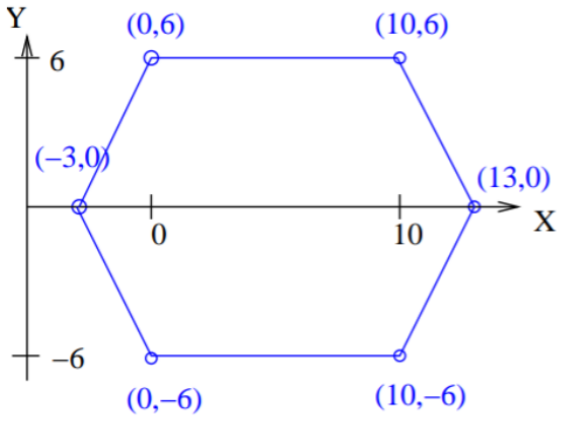


**Figure 2.** Top down view of the room polygons

Each polygon was provided with a color and a material. The colors are a mix of subtle white colors and bright colorful

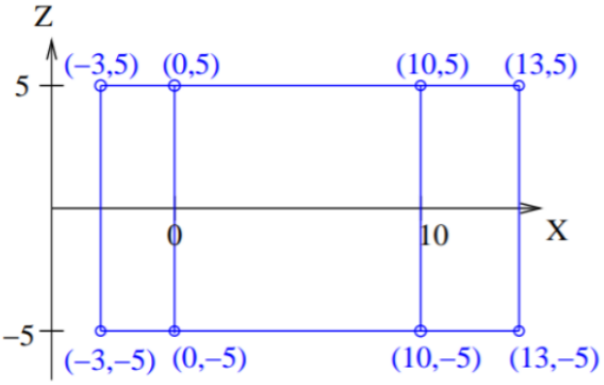


**Figure 3.** Side view of the room polygons

ones, to show color bleeding interactions between the polygons. The material could be diffuse, reflective or refractive to showcase the capabilities of the model. Additionally objects were placed into the room, two spheres and a tetrahedron. The objects each showcased one of the three available materials. A light source in the form of an area light was then placed on the ”roof” of the room, lighting up the scene from above.

### 3.2 Rays

The ray was described using the parametric equation

$$\vec{R}(t) = \vec{O} + t\vec{D} \tag{9}$$

Where:
$\vec{R}(t)$ is the point on the ray $\vec{O}$ is the ray origin (starting point) $t$ is the parameter (distance along the ray) $\vec{D}$ is the ray direction vector.

The ray class was also equipped with an importance value and a pointer to keep track of the radiance and the path tracing through out its lifetime. The size of the desired output determined how many initial rays should be cast into the scene, since for every pixel a ray was cast into the scene.

**Ray intersections** were checked by going through all the available shapes in the scene and finding the closest intersection. Then the properties of that shape, such as material and color were gathered to determine the new path and added radiance of the ray. Once the ray has reached the base case for termination it returns it’s radiance as a color value. This process was then repeated for every pixel.

### 3.3 Illumination

To calculate whether an object is lit by the light source, shadow-rays were cast from the hit point onto random points on the light source. If the distance to the light from the object is smaller than the distance to the intersecting random point on the light source, the object is in shadow and will not receive any light contribution. If the distance requirement is fulfilled the object has a light contribution on the given point. This contribution was calculated by an estimator[2]:

$$(L_D(x \to \theta_0)) = \frac{A_L e}{N} \sum_{i=1}^{N} f(x, d_i, \theta_0) \frac{\cos \Omega_{x,i} \cos \Omega_{y,i}}{||d_i||^2} \quad (10)$$

where $E(x)$ is the expected value $A_L e$ is the effective area of the light source $N$ is the number of rays/samples $\Omega_{x,i}$ and $\Omega_{y,i}$ are the azimuth and elevation angles for the $i$-th ray $d_i$ is the direction vector for the $i$-th ray $N_x$ and $N_y$ are the normals of the point on the light source and the object.

**Indirect light** was calculated if the intersecting object had the diffuse material property. The algorithm generates a number of random samples within the hemisphere around the surface normal. This is done to approximate the integration of the diffuse reflection. For each sample, the algorithm generates a random direction in spherical coordinates ($\theta$ and $\phi$) and then converts it to Cartesian coordinates $(x, y, z)$ using the following equations:

$$x = \sin(\phi)\cos(\theta)\ y = \sin(\phi)\sin(\theta)\ z = \cos(\phi) \quad (11)$$

For each random direction, the algorithm creates a new ray starting from a slightly offset point on the surface (to avoid self-intersection) and with the transformed random direction. This new ray is then linked to the original ray as the next ray to be traced. The algorithm recursively traces each of the newly created rays and accumulates the resulting color. The contribution of each sample is weighted by the cosine of the angle between the random direction and the surface normal using the BRDF. For a diffuse object this light contribution is summed with the direct light contribution.

### 3.4 Reflections and refractions

Two spheres were placed in the scene, one with a perfect reflective material (mirror) and one with a refractive material (glass). The mirror material is calculated by using the law of reflection 5 and uses recursion to generate a new ray with the given direction from the equation. The refractive rays are calculated by using Snell's law and their direction was calculated as:

$$\vec{t} = n\vec{d} + (n\cos\theta_i - \cos\theta_t)\vec{n} \quad (12)$$

The algorithm the created two new rays, one for the reflection and one for the refraction, with their respective directions and starting from slightly offset points on the surface to avoid self-intersection. Then the paths of the reflection and refraction rays were traced and the resulting color is blended according to the Fresnel reflection coefficient $F_r$. The final color is the sum of the reflected color multiplied by $F_r$ and the refracted color multiplied by $(1 - F_r)$.

## 4. Result

The result of the renderer is a 600x600 image. By varying the number of samples, the number of shadow rays per pixel, and the depth parameter, how many bounces were allowed, different results were achieved.

Results of varying the number of direct light samples:

| Figure | Direct light | Indirect light | Render time |
|---|---|---|---|
| 4 | 8 | 8 | 120.93s |
| 5 | 16 | 8 | 208.80s |
| 6 | 64 | 8 | 751.50s |

**Table 1.** The result of varying the number of samples.

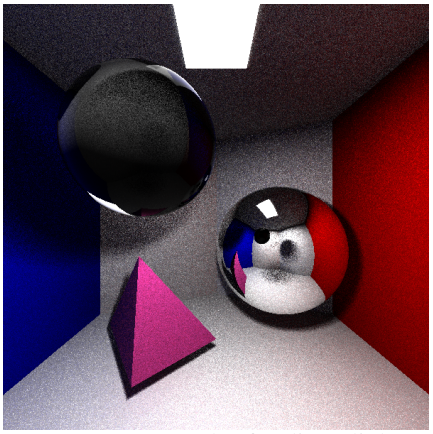

**Figure 4.** 8 samples for direct light.

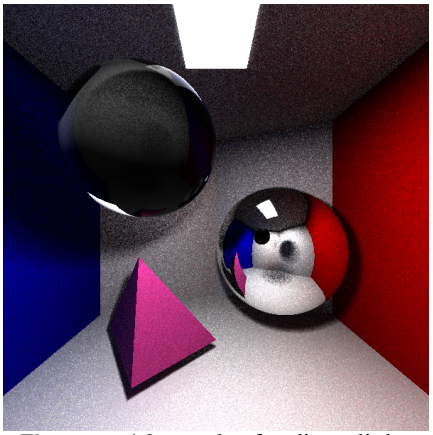

**Figure 5.** 16 samples for direct light.

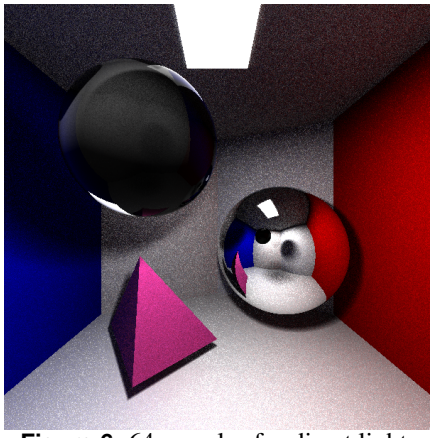

**Figure 6.** 64 samples for direct light.

Renders with varying depth value and indirect at 8 samples:

| Figure | Direct light | Depth | Render time |
|---|---|---|---|
| 7 | 16 | 1 | 27.97s |
| 8 | 16 | 2 | 208.80s |
| 9 | 16 | 3 | 1554.18s |

**Table 2.** The result of varying the number of samples.

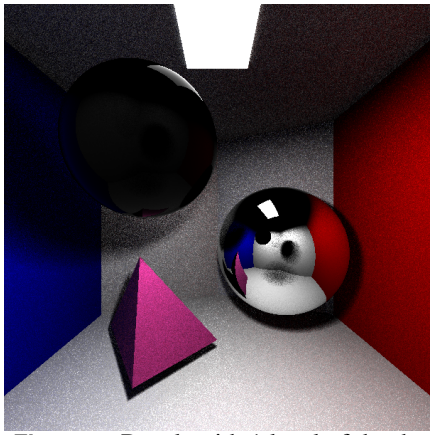

**Figure 7.** Result with 1 level of depth.

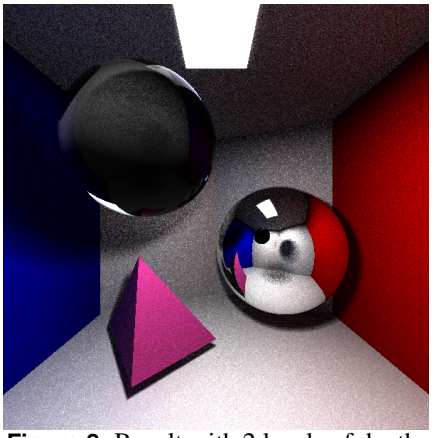

**Figure 8.** Result with 2 levels of depth.

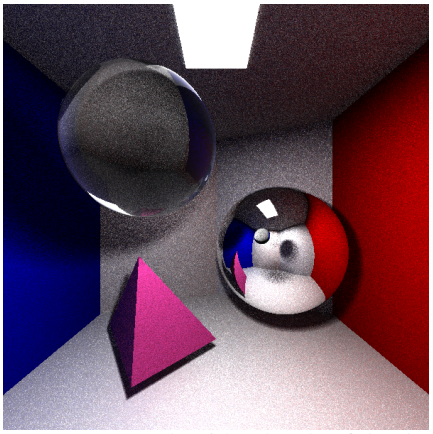

**Figure 9.** Result with 3 levels of depth.

## 5. Discussion

The ray tracing algorithm implemented in this project was able to accurately simulate complex global illumination effects such as color bleeding, reflections, and refractions. The use of multiple importance sampling for both direct and indirect lighting contributed to the realistic appearance of the rendered scenes.

### 5.1 Sampling and Image Quality

The results demonstrate a correlation between the number of samples and image quality, particularly in shadow regions. As shown in Figures 4, 5 and 6, increasing the number of direct light samples from 8 to 64 resulted in reduced noise in the rendered images. Altough it might be hard to see in the figures, it can be perceived by looking at shadows specifically. Since the direct light samples control the number of shadow rays that spawn, a lower number will result in noisy and sharp shadows. However, the quality improvement comes at a significant computational cost, with render times increasing from 120.93s to 751.50s.

### 5.2 Depth

The depth, or the number of times a ray is allowed to bounce before terminating, plays a large role i creating a realistic

image. The more depth levels the more bounces and thus the more realism. Figures 7, 8 and 9 demonstrate that an increasing depth level enables more interesting results. With only one level of depth seen in figure 7, the scene appears notably darker and lacks the subtle indirect illumination effects that make ray-traced images realistic. Specifically the roof and the glass material is impacted, since they rely the most on indirect light contribution. The most striking difference is observed in the glass sphere's caustics, which are almost not visible at depth 1 but become progressively more visible at depths 2 and 3. However, the computational cost increases exponentially with depth, as shown by table 2 the render times jumping from 27.97s at depth 1 to 1554.18s at depth 3. This exponential growth occurs because each bounce potentially spawns multiple new rays that must be traced through the scene. For example with a depth of 3 and 16 samples for direct light rendered at an output of 600x600. The amount of rays become 25 for the first layer, 192 for the second and 1024 for the third, totaling up to 1241 for one pixel, multiplied with the number of total pixels 1241*600*600 = 446,760,000 rays. This explains the why the render time increases at such a exponential rate. As the samples increase so does the color bleed from the walls to floor as well, note the subtle change of tint on the floor as the bleed becomes more apparent the more depth is used.

### 5.3 Improvements

Some improvements that can be explored are:

- Adaptive sampling - Vary the number of samples per pixel based on scene content. Rather than using the same number of samples everywhere. Areas with high variance (like sharp edges, complex reflections, or noisy areas) receive more samples and more simple areas receive fewer.
- Denoising techniques - This is an post-process step that would include passing the image through a filter to remove the noise created by the monte-carlo integration.
- Anti-Aliasing - Since jagged lines are visible throughout the image specifically on objects with sharp edges, such as the tetrahedron. An anti-alias technique could be utilized to mitigate this.

## 6. Conclusion

In conclusion, the custom ray-tracing model implemented in this project effectively demonstrates the power of global illumination techniques for achieving photorealistic rendering. By leveraging Monte Carlo integration and importance sampling, the renderer produces realistic lighting effects, including accurate reflections, refractions, and color bleeding. However, the project also highlights the significant computational costs associated with increasing sample rates and ray depths, which enhance visual realism at the expense of longer render times. Future optimizations, such as adaptive sampling, denoising, and anti-aliasing, could improve both efficiency and image quality.

## References

[1] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, 1980.

[2] Mark Eric Dieckmann. Tncg15: Advanced global illumination and rendering. Lecture slides from Linköping University, Campus Norrköping, 2024. Lecture 1–9.