3D Ray Tracing Engine

Project Documentation

Visual and Analytical Computing
University of Rostock
Winter Semester 2025/26

Authors:

Abu Bakar
M Shahman Butt

Supervisor:

Professor Olivier Staadt

February 2026

Contents

1 Introduction

This document presents the technical documentation for our Monte Carlo Path Tracing engine, developed as part of the Visual and Analytical Computing course at the University of Rostock. The project spans approximately four months of development, resulting in a fully functional physically-based renderer with an interactive graphical interface.

Our primary objective was to build a ray tracer from first principles, implementing the core mathematical foundations ourselves rather than relying on existing rendering libraries. This approach gave us deep insight into how light simulation works at a fundamental level and the computational challenges involved in producing realistic images.

The engine handles scenes with complex geometry, multiple material types, and various lighting conditions. We focused particularly on implementing efficient acceleration structures and variance reduction techniques to make the renderer practical for scenes with tens of thousands of triangles.

2 Project Overview

2.1 Motivation

Ray tracing has become increasingly relevant in modern graphics, with real-time implementations now appearing in consumer hardware. Understanding the underlying principles is essential for anyone working in computer graphics, game development, or visual effects.

We chose to implement a path tracer specifically because it naturally handles global illumination effects like soft shadows, color bleeding, and caustics. These phenomena are difficult or impossible to achieve with traditional rasterization approaches.

2.2 Scope

The project covers the following areas:

- Core ray-scene intersection algorithms
- Physically-based material models (Lambertian, metallic, dielectric, GGX)
- Spatial acceleration using Bounding Volume Hierarchies
- Monte Carlo integration with importance sampling
- Multiple Importance Sampling for variance reduction
- Multi-threaded rendering with tile-based work distribution
- AI-based denoising integration
- Interactive GUI for real-time scene exploration

2.3 Task Distribution

The workload was divided between two team members:

Abu Bakar	**M Shahman Butt**
Core engine architecture	Material system (GGX, metals)
BVH implementation	MIS and NEE sampling
XML scene parsing	GUI development
OBJ/MTL file loading	Camera controls
Multi-threading system	Denoiser integration

Table 1: Task distribution between team members

3 Mathematical Foundations

3.1 The Rendering Equation

The theoretical basis of our renderer is Kajiya's rendering equation, formulated in 1986. It describes how light propagates through a scene:

$$L_o(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + \int_\Omega f_r(\mathbf{x}, \omega_i, \omega_o) L_i(\mathbf{x}, \omega_i)(\omega_i \cdot \mathbf{n})\, d\omega_i \tag{1}$$

Here, L_o is the outgoing radiance at point $\mathbf{x}$ in direction ω_o, L_e is the emitted radiance, f_r is the Bidirectional Reflectance Distribution Function (BRDF), and the integral sums contributions from all incoming directions over the hemisphere Ω.

This equation is recursive—the incoming radiance L_i at one point depends on the outgoing radiance at another point. Solving it analytically is impossible for non-trivial scenes, which is why we turn to Monte Carlo methods.

3.2 Monte Carlo Integration

Monte Carlo integration estimates integrals by random sampling:

$$\int f(x)\, dx \approx \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)} \tag{2}$$

where x_i are samples drawn from probability distribution $p(x)$. The key insight is that we can choose $p(x)$ to concentrate samples where $f(x)$ is large, reducing variance. This technique, called importance sampling, is crucial for efficient rendering.

In our implementation, we sample directions according to the BRDF rather than uniformly over the hemisphere. For a Lambertian surface, this means sampling proportionally to $\cos\theta$, which matches the geometry term in the rendering equation.

3.3 Path Tracing Algorithm

Path tracing simulates light by tracing rays from the camera into the scene. When a ray hits a surface, it spawns a new ray based on the material properties. This continues until the ray either hits a light source, escapes the scene, or is terminated.

Listing 1: Simplified path tracing loop

```
color trace_ray(ray r, int depth) {
    if (depth <= 0) return black;

```

```
    hit_record rec;
    if (!scene.hit(r, rec))
        return background_color;

    color emitted = rec.material->emit();

    scatter_record srec;
    if (!rec.material->scatter(r, rec, srec))
        return emitted;

    ray scattered = srec.scattered_ray;
    color attenuation = srec.attenuation;

    return emitted + attenuation * trace_ray(scattered, depth -
       1);
}
```

The challenge is that basic path tracing can be extremely noisy, especially for small light sources. We address this through the sampling techniques described in Section 7.

4 Ray-Scene Intersection

4.1 Ray Representation

A ray is defined parametrically as:

$$\mathbf{P}(t) = \mathbf{O} + t \cdot \mathbf{D} \tag{3}$$

where $\mathbf{O}$ is the origin, $\mathbf{D}$ is the normalized direction, and $t \geq 0$ is the parameter. Finding intersections means solving for the smallest positive t where the ray intersects scene geometry.

4.2 Sphere Intersection

For a sphere centered at $\mathbf{C}$ with radius r, the intersection is found by substituting the ray equation into the sphere equation $|\mathbf{P} - \mathbf{C}|^2 = r^2$:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{4}$$

where $a = \mathbf{D} \cdot \mathbf{D}$, $b = 2\mathbf{D} \cdot (\mathbf{O} - \mathbf{C})$, and $c = |\mathbf{O} - \mathbf{C}|^2 - r^2$.

4.3 Triangle Intersection: Möller-Trumbore

For mesh geometry, we use the Möller-Trumbore algorithm, which is efficient because it avoids computing the plane equation explicitly. Given a triangle with vertices $\mathbf{V}_0$, $\mathbf{V}_1$, $\mathbf{V}_2$, the algorithm solves:

$$\mathbf{O} + t\mathbf{D} = (1 - u - v)\mathbf{V}_0 + u\mathbf{V}_1 + v\mathbf{V}_2 \tag{5}$$

The parameters (u, v) are barycentric coordinates used for texture mapping and normal interpolation. Our implementation handles back-face culling and ensures numerical stability for grazing angles.

4.4 Quad Primitive

We implemented a quad primitive for area lights and flat surfaces like walls. A quad is defined by a corner point **Q** and two edge vectors **u** and **v**. The intersection test first finds where the ray hits the infinite plane, then checks if the hit point lies within the parallelogram bounds.

This is more efficient than splitting quads into two triangles and allows direct sampling for importance sampling of area lights.

5 Bounding Volume Hierarchy

5.1 The Need for Acceleration

Testing every ray against every triangle has $O(n)$ complexity per ray. For a scene with 100,000 triangles and an image with 1 million pixels at 100 samples each, this means 10^{13} intersection tests—completely impractical.

A Bounding Volume Hierarchy (BVH) organizes geometry into a tree structure where each node contains a bounding box. If a ray misses a node's bounding box, all geometry inside can be skipped. This reduces average complexity to $O(\log n)$.

5.2 Axis-Aligned Bounding Boxes

We use Axis-Aligned Bounding Boxes (AABBs) because the intersection test is extremely fast. An AABB is defined by minimum and maximum corners. The ray-AABB test uses the slab method:

Listing 2: Ray-AABB intersection

```
bool aabb::hit(const ray& r, double t_min, double t_max) {
    for (int axis = 0; axis < 3; axis++) {
        double inv_d = 1.0 / r.direction()[axis];
        double t0 = (min[axis] - r.origin()[axis]) * inv_d;
        double t1 = (max[axis] - r.origin()[axis]) * inv_d;
        if (inv_d < 0) std::swap(t0, t1);
        t_min = std::max(t0, t_min);
        t_max = std::min(t1, t_max);
        if (t_max <= t_min) return false;
    }
    return true;
}
```

5.3 Surface Area Heuristic

The quality of a BVH depends heavily on how geometry is partitioned. We use the Surface Area Heuristic (SAH), which estimates the expected cost of traversing a subtree.

The SAH cost for splitting a node into children A and B is:

$$C = C_{\text{trav}} + \frac{SA(A)}{SA(N)} \cdot N_A \cdot C_{\text{isect}} + \frac{SA(B)}{SA(N)} \cdot N_B \cdot C_{\text{isect}} \tag{6}$$

where SA is surface area, N_A and N_B are primitive counts, C_{trav} is traversal cost, and C_{isect} is intersection cost. We evaluate splits along each axis and choose the one with minimum cost.

In practice, SAH produces significantly better trees than simpler strategies like object median or spatial median splitting. Our benchmarks showed 2-3x speedup on the Stanford Bunny mesh compared to median splitting.

6 Material System

6.1 BRDF Overview

The Bidirectional Reflectance Distribution Function describes how light reflects off a surface. It takes an incoming direction ω_i and outgoing direction ω_o and returns the ratio of reflected radiance to incident irradiance.

A physically plausible BRDF must satisfy two properties:

1. **Reciprocity**: $f_r(\omega_i, \omega_o) = f_r(\omega_o, \omega_i)$
2. **Energy conservation**: $\int f_r(\omega_i, \omega_o) \cos\theta_o \, d\omega_o \leq 1$

6.2 Lambertian Diffuse

The simplest BRDF is Lambertian, which scatters light equally in all directions:

$$f_r = \frac{\rho}{\pi} \tag{7}$$

where ρ is the albedo (surface color). The π factor ensures energy conservation. We sample Lambertian surfaces using cosine-weighted hemisphere sampling, which matches the $\cos\theta$ factor in the rendering equation and reduces variance.

6.3 Perfect Specular Reflection

Mirrors reflect light in a single direction: $\omega_r = 2(\omega_i \cdot \mathbf{n})\mathbf{n} - \omega_i$. This is technically a delta distribution in the BRDF, requiring special handling in the path tracer.

For rough metals, we add a "fuzz" parameter that randomly perturbs the reflected direction, creating blurred reflections.

6.4 Dielectric Materials

Glass and water both reflect and refract light. The split is governed by the Fresnel equations, which we approximate using Schlick's formula:

$$R(\theta) = R_0 + (1 - R_0)(1 - \cos\theta)^5 \tag{8}$$

where $R_0 = \left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2$ and n_1, n_2 are refractive indices.

Refraction direction is computed using Snell's law. Total internal reflection occurs when the angle is too steep for light to escape a denser medium.

6.5 GGX Microfacet Model

For realistic metallic and plastic surfaces, we implemented the GGX microfacet model. The core idea is that real surfaces consist of tiny mirror-like facets with varying orientations. The roughness parameter controls how spread out these orientations are.

The GGX normal distribution function is:

$$D(\mathbf{h}) = \frac{\alpha^2}{\pi((\mathbf{n} \cdot \mathbf{h})^2(\alpha^2 - 1) + 1)^2} \tag{9}$$

where $\mathbf{h}$ is the half-vector and α is roughness. Low α gives sharp highlights; high α gives diffuse-like appearance.

The geometry term G accounts for microfacet shadowing and masking. We use the Smith separable form with GGX:

$$G(\omega_i, \omega_o) = G_1(\omega_i) \cdot G_1(\omega_o) \tag{10}$$

where $G_1(\omega) = \frac{2(\mathbf{n}\cdot\omega)}{(\mathbf{n}\cdot\omega)+\sqrt{\alpha^2+(1-\alpha^2)(\mathbf{n}\cdot\omega)^2}}$.

7 Importance Sampling Techniques

7.1 PDF Classes

Our sampling system is built around an abstract `pdf` class with methods `value()` (evaluate PDF at a direction) and `generate()` (sample a random direction). Concrete implementations include:

- `cosine_pdf`: Samples proportionally to $\cos\theta$ for Lambertian surfaces
- `hittable_pdf`: Samples toward a specific object (used for light sources)
- `mixture_pdf`: Combines two PDFs with 50/50 weighting

7.2 Next Event Estimation

Standard path tracing struggles with small light sources because rays rarely hit them by chance. Next Event Estimation (NEE) solves this by explicitly sampling the light at each bounce.

At each intersection, we:

1. Pick a random point on a light source
2. Trace a shadow ray to check visibility
3. If unoccluded, add the direct illumination contribution

This dramatically reduces variance for scenes with small or distant lights.

7.3 Multiple Importance Sampling

A problem arises when combining NEE with standard path tracing: some light paths get counted twice. Multiple Importance Sampling (MIS) solves this by weighting contributions from different sampling strategies.

We use the power heuristic:

$$w_s = \frac{p_s^\beta}{\sum_k p_k^\beta} \tag{11}$$

with $\beta = 2$. Each sample is weighted by how "good" its strategy was for that particular path. This ensures that the strategy best suited for each situation contributes most.

7.4 Russian Roulette

Naive path termination after a fixed number of bounces introduces bias (missing energy from longer paths). Russian Roulette randomly terminates paths while compensating for the lost energy:

Listing 3: Russian Roulette termination

```
double p_continue = std::min(0.95, luminance(throughput));
if (random_double() > p_continue)
    return black;
throughput /= p_continue;  // Compensate for termination
```

This produces an unbiased estimator while allowing early termination of low-energy paths.

8 System Architecture

8.1 Module Structure

The codebase is organized into distinct modules:

- **engine/**: Core rendering logic, materials, geometry
- **gui/**: Interactive application using Dear ImGui
- **3rdParty/**: External libraries (tinyxml2, stb_image, GLFW)

This separation allows the command-line renderer and GUI application to share the same engine code.

8.2 Scene Loading

Scenes are defined in XML format, specifying camera parameters, materials, and object placements. The parser supports:

- Primitive shapes (spheres, quads)
- OBJ mesh loading with MTL material files
- Material properties (albedo, roughness, emission)
- Transformations (position, rotation, scale)

OBJ files can contain hundreds of thousands of triangles. Our loader handles this efficiently and automatically builds the BVH during scene initialization.

8.3 Multi-threaded Rendering

We use tile-based rendering to parallelize across CPU cores. The image is divided into 32×32 pixel tiles, and worker threads process tiles from a shared queue.

This approach has several advantages:

- **Load balancing**: Complex regions (e.g., glass) take longer per tile; idle threads pick up remaining work
- **Cache efficiency**: Working on adjacent pixels keeps relevant BVH nodes in cache
- **Progressive display**: Tiles can be displayed as they complete

Thread synchronization uses atomic operations and mutex-protected queues to avoid contention.

9 Interactive Application

9.1 GUI Framework

The interactive renderer uses Dear ImGui for the interface and GLFW/OpenGL for window management. The rendered image is uploaded to a GPU texture for display.

Controls include:

- WASD keys for camera movement
- Mouse drag for camera rotation
- Sliders for samples, resolution, material parameters
- Scene file selection and hot-reloading

9.2 Progressive Rendering

Rather than waiting for a full render, the GUI displays partial results. Each frame accumulates more samples, and the image gradually converges to the final result.

Moving the camera resets the sample count, providing instant (though noisy) feedback. This makes scene exploration interactive despite the computational cost of path tracing.

9.3 Denoiser Integration

We integrated Intel's Open Image Denoise (OIDN) library to remove noise from low-sample-count renders. OIDN uses machine learning trained on path-traced images to predict the noise-free result.

The denoiser runs as a post-process after rendering completes. It takes the RGB image as input and outputs a denoised version. This allows acceptable quality at 50-100 samples instead of the thousands typically needed.

10 Results and Performance

10.1 Test Scenes

We validated the renderer on several test scenes:

- **Cornell Box**: Classic test for global illumination, color bleeding
- **Cornell Box with Water**: Tests refraction and caustics

- **Stanford Bunny**: High-poly mesh (69,451 triangles) for BVH testing
- **100 Spheres**: Stress test with multiple materials

10.2 Performance Measurements

Benchmarks were run on an Apple M1 MacBook Pro:

Scene	Triangles	Linear	BVH
Cornell Box	36	2.1s	1.8s
100 Spheres	0 (spheres)	4.2s	3.8s
Stanford Bunny	69,451	847s	12.3s
Dragon	871,414	DNF	89s

Table 2: Render time comparison at 800×600, 100 samples

The BVH provides massive speedups for mesh-heavy scenes. The Dragon model would take over 10 hours with linear traversal but renders in under 2 minutes with BVH.

10.3 Image Quality

The implemented techniques produce physically plausible results:

- Soft shadows from area lights
- Color bleeding between surfaces
- Accurate glass refraction with Fresnel effects
- Metallic highlights with configurable roughness

11 Challenges Encountered

11.1 Numerical Precision

Floating-point errors caused "shadow acne" where surfaces shadowed themselves due to imprecise intersection calculations. The solution was adding a small epsilon offset when spawning shadow rays.

11.2 BVH Debugging

Incorrect BVH construction caused rays to miss geometry entirely. We added visualization tools to display bounding boxes and verify the tree structure. The issue was an off-by-one error in the SAH binning code.

11.3 Energy Conservation

Early material implementations violated energy conservation, causing scenes to "blow out" after many bounces. We traced this to missing normalization factors in the BRDF calculations.

11.4 Thread Safety

Race conditions in the tile queue caused duplicate work and missing tiles. Proper mutex usage and atomic operations resolved these issues, though debugging was time-consuming.

12 Future Work

Several extensions would improve the renderer:

- **GPU acceleration**: CUDA or Metal compute shaders would provide 10-100x speedup
- **Volumetric rendering**: Fog, smoke, and subsurface scattering
- **Texture mapping**: Image-based albedo, normal, and roughness maps
- **Environment maps**: HDR lighting from panoramic images
- **Bidirectional path tracing**: Better handling of caustics

13 Conclusion

This project successfully implemented a functional Monte Carlo path tracer with physically-based materials and efficient acceleration structures. The combination of BVH traversal, importance sampling, and AI denoising makes it practical for scenes with complex geometry.

Building the renderer from scratch provided deep understanding of light transport theory and the numerical techniques required for practical implementation. The interactive GUI adds educational value by allowing real-time exploration of rendering parameters.

The codebase serves as a foundation for further experimentation with advanced rendering techniques.

Repository: `https://github.com/ab17dogar/project-csi`