Contents

Project Specification Report

3D Ray Tracing Engine for Photorealistic Image Generation

Course: Project CSI in Visual Computing
Program: Masters in Computer Science International (CSI)
Institution: Universität Rostock
Student: [Your Name]

---

1. Project Overview

1.1 Project Description

This project aims to develop a comprehensive 3D ray tracing engine from scratch that generates photorealistic images of complex 3D scenes. The engine will accept scene descriptions in XML format, 3D mesh models in OBJ format, and material definitions in MTL format, producing high-quality rendered images through physically-based rendering techniques.

Ray tracing is a fundamental computer graphics technique that simulates the physical behavior of light by tracing rays from the camera through each pixel into the scene, calculating interactions with surfaces, materials, and light sources. Unlike rasterization-based approaches, ray tracing naturally handles complex lighting phenomena including reflections, refractions, soft shadows, and global illumination, making it the gold standard for photorealistic rendering.

1.2 Objectives

- Develop a complete CPU-based ray tracing engine implementing core rendering algorithms
- Support multiple input formats (XML scene files, OBJ meshes, MTL materials)
- Implement advanced material models and lighting techniques
- Achieve high-quality photorealistic output comparable to industry standards
- Optimize performance through multi-threading and spatial acceleration structures

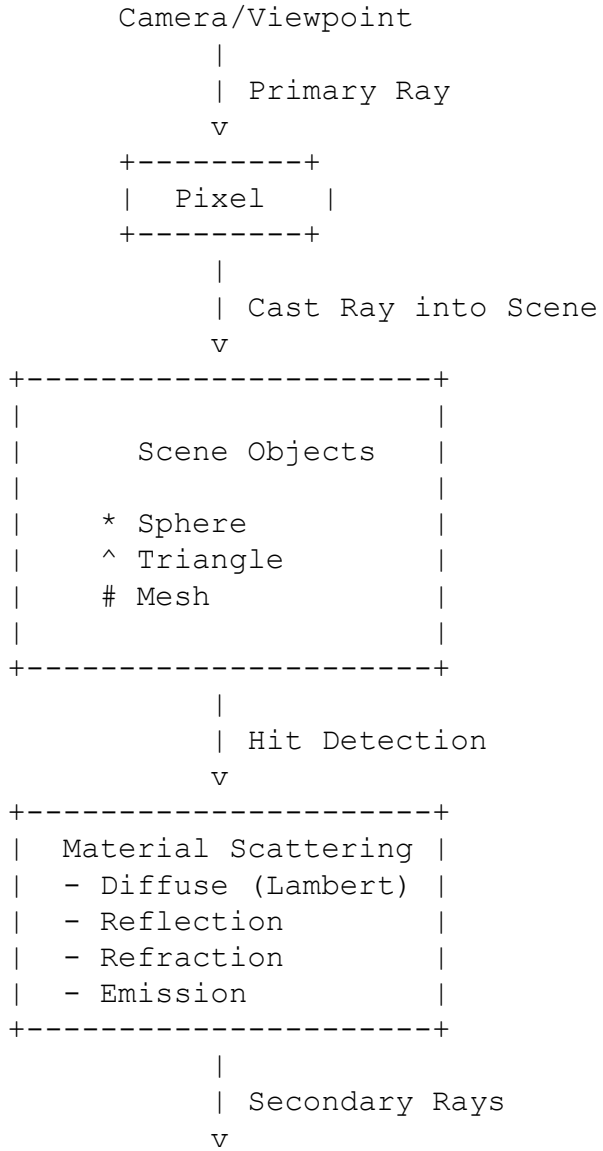
- Explore GPU acceleration possibilities for real-time rendering

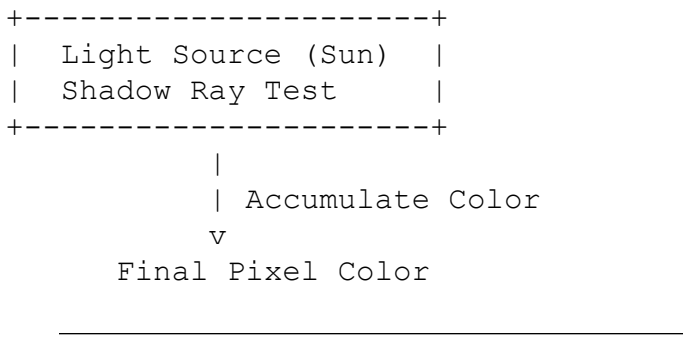
1.3 Motivation and Justification

Ray tracing has become increasingly important in computer graphics, with modern GPUs now supporting hardware-accelerated ray tracing (RTX, RDNA2). Understanding the fundamental algorithms and implementation details is crucial for computer graphics professionals. This project provides hands-on experience with:

- **Core Algorithms:** Ray-object intersection, material scattering models, recursive ray tracing
- **Performance Optimization:** Spatial data structures (BVH, KD-trees), multi-threading, SIMD optimizations
- **Advanced Techniques:** Monte Carlo path tracing, importance sampling, denoising algorithms
- **System Design:** Modular architecture, extensible material system, efficient memory management

1.4 Ray Tracing Concept Visualization

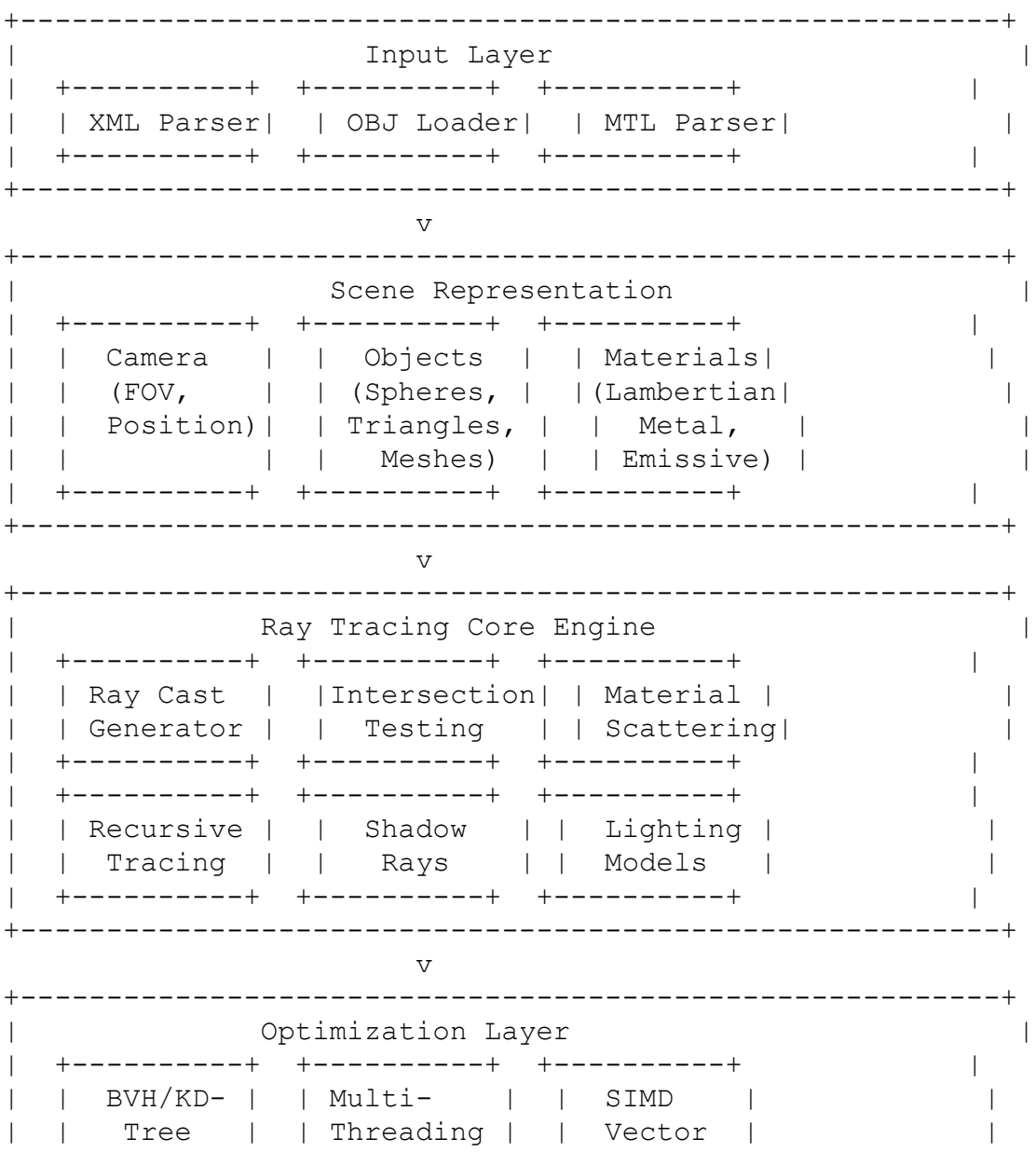


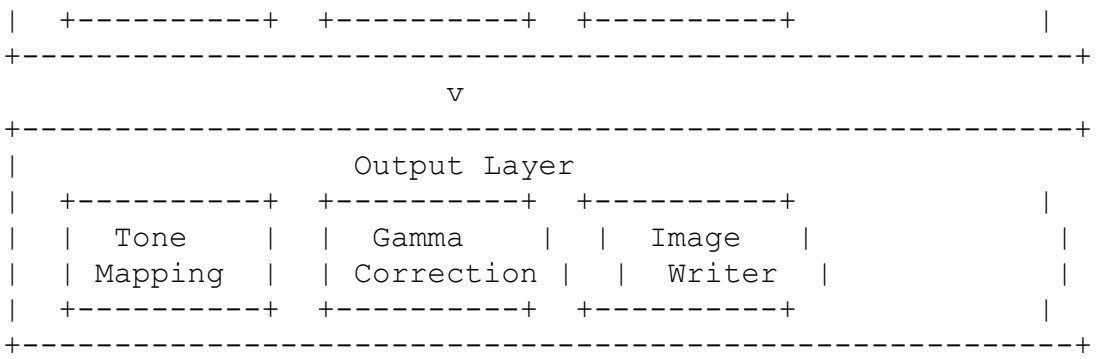


---

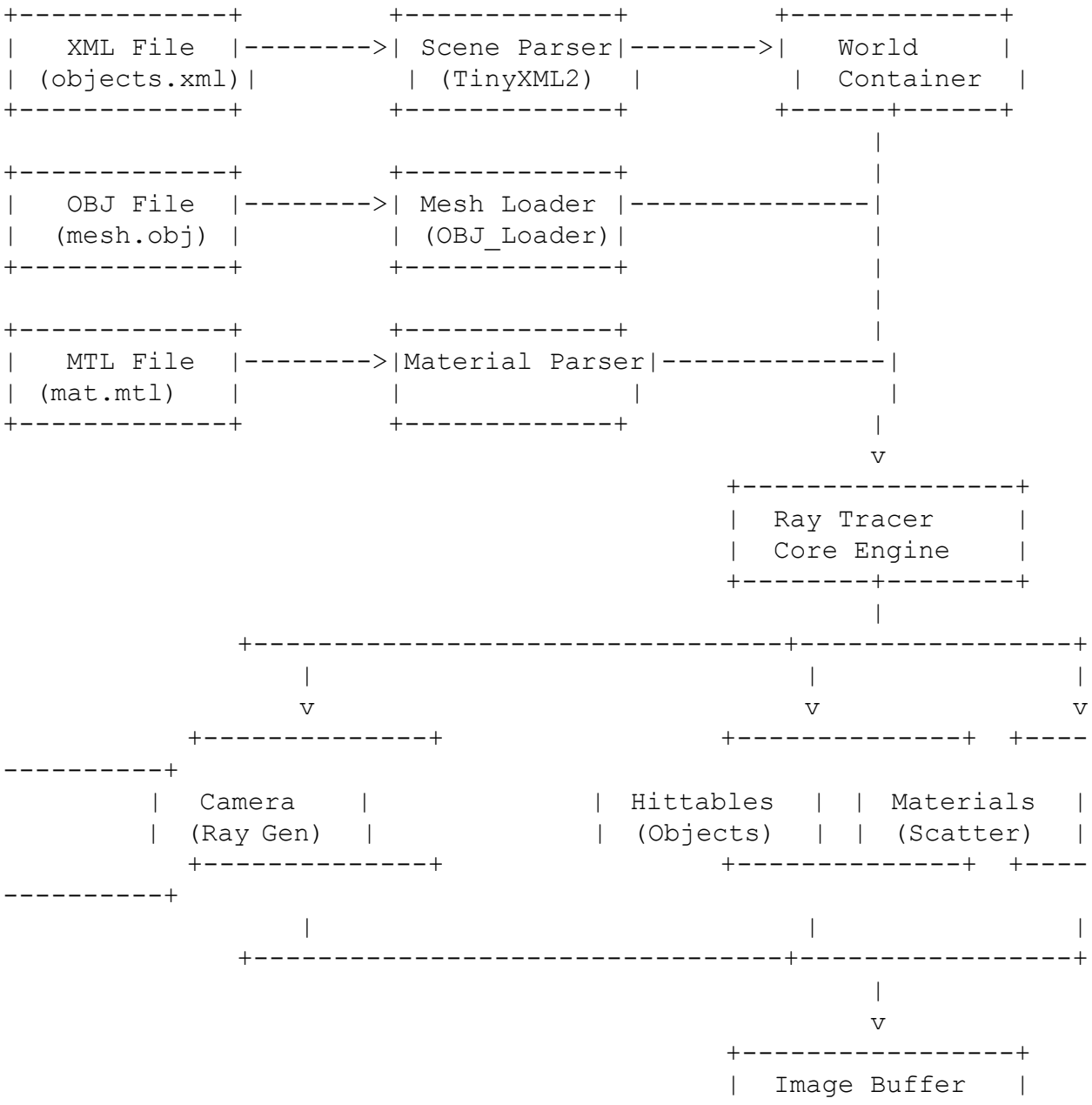
2. System Architecture and Design

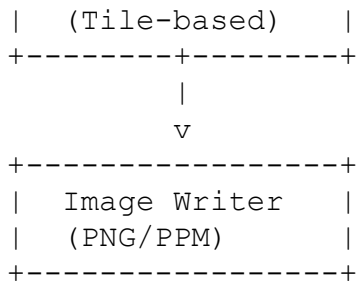
2.1 High-Level Architecture





2.2 Component Interaction Diagram





2.2 Design Decisions and Justifications

2.2.1 Object-Oriented Design

- **Decision:** Use polymorphic base classes (`hittable`, `material`) with virtual functions
- **Justification:** Enables extensibility - new primitives and materials can be added without modifying core rendering loop. Follows Open/Closed Principle.

2.2.2 XML Scene Description

- **Decision:** Use XML format for scene configuration (camera, lights, objects, materials)
- **Justification:** Human-readable, hierarchical structure, easy to parse and validate. Industry-standard approach (similar to POV-Ray, Blender).

2.2.3 Tile-Based Parallel Rendering

- **Decision:** Divide image into tiles processed by separate threads
- **Justification:** Maximizes cache locality, enables progress tracking, scales well with CPU cores. Reduces memory contention compared to pixel-level parallelism.

2.2.4 Recursive Ray Tracing

- **Decision:** Implement recursive ray tracing with configurable depth limit
- **Justification:** Natural implementation of reflections and refractions. Depth limit prevents infinite recursion and controls quality/performance tradeoff.

2.2.5 Material System Design

- **Decision:** Separate material properties from geometry, use scatter functions
- **Justification:** Physically-based rendering model. Materials define how light interacts (absorption, reflection, emission) independently of object shape.

---

3. Core Algorithms and Techniques

3.0 Ray Tracing Algorithm Flow

Complete Ray Tracing Pipeline:

```
START: For each pixel in image
|
```

```
+-> Generate Primary Ray from Camera
    |
    +-> Ray = Camera.origin + (u,v) * Camera.direction
    |
    +-> Test Intersection with Scene Objects
        |
        +-> [BVH Traversal] Find closest intersection
        |   |
        |   +-> Test Ray vs Bounding Boxes
        |   |
        |   +-> Test Ray vs Objects in hit nodes
        |
        +-> If Hit Found:
            |
            +-> Calculate Hit Point & Normal
            |
            +-> Get Material Properties
            |
            +-> [Material Scattering]
            |   |
            |   +-> Diffuse: Random hemisphere direction
            |   +-> Metal: Reflect ray
            |   +-> Glass: Refract + Reflect (Fresnel)
            |   +-> Emissive: Add light color
            |
            +-> [Recursive Tracing] (if depth > 0)
            |   |
            |   +-> Cast Secondary Ray
            |       +-> Recursively call ray_color()
            |
            +-> [Shadow Testing]
                |
                +-> Cast Shadow Ray to Light
                    +-> If occluded: Darken color

+-> Accumulate Color (Monte Carlo: average samples)
    |
    +-> Apply Tone Mapping & Gamma Correction
        |
        +-> Write to Image Buffer
            |
            +-> END: Save Image File
```

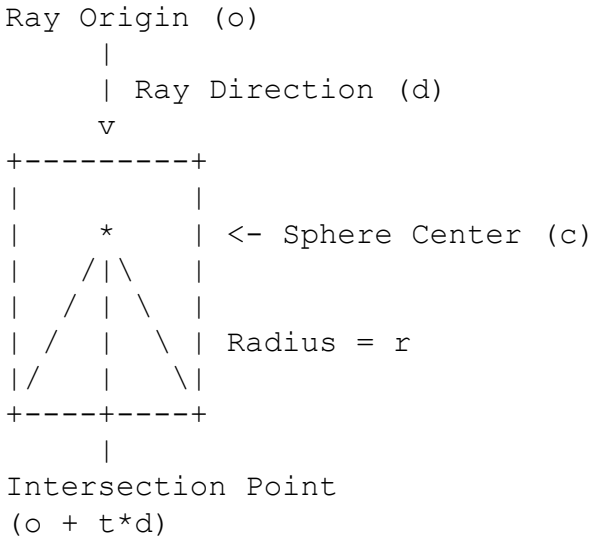
3.1 Ray-Object Intersection Algorithms

3.1.1 Ray-Sphere Intersection

- **Algorithm:** Analytic solution using quadratic equation

- **Formula:** $t^2(d \cdot d) + 2t(d \cdot (o-c)) + (o-c) \cdot (o-c) - r^2 = 0$
- **Complexity:** O(1) per sphere

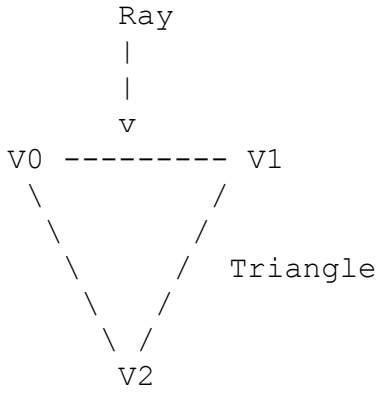
Visualization:



3.1.2 Ray-Triangle Intersection

- **Algorithm:** Möller-Trumbore algorithm
- **Advantages:** Fast, numerically stable, computes barycentric coordinates
- **Complexity:** O(1) per triangle

Visualization:




```
Intersection: Ray hits triangle plane
Barycentric: (u, v, w) where u+v+w=1
Point = u*V0 + v*V1 + w*V2
```

3.1.3 Ray-Mesh Intersection

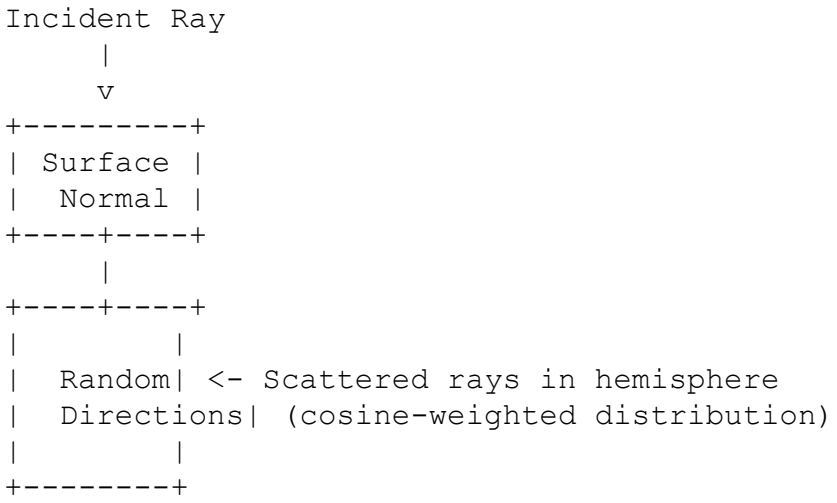
- **Algorithm:** Test ray against all triangles in mesh
- **Optimization:** Spatial acceleration structures (BVH) for complex meshes
- **Complexity:** O(log n) with BVH, O(n) without

3.2 Material Scattering Models

3.2.1 Lambertian (Diffuse) Material

- **Model:** Perfectly diffuse surface, cosine-weighted hemisphere sampling
- **Implementation:** Random direction in hemisphere around surface normal
- **Use Case:** Matte surfaces (paper, unpolished wood, fabric)

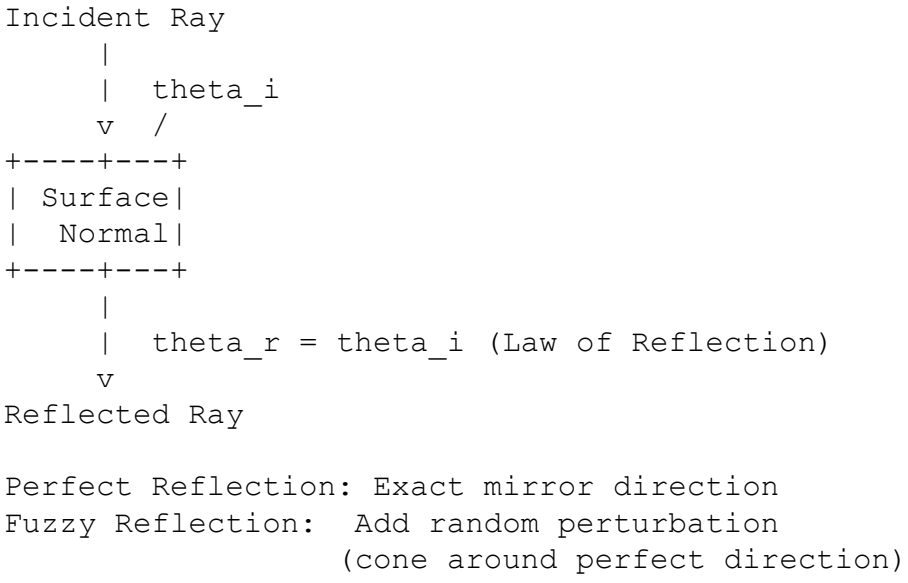
Scattering Visualization:



3.2.2 Metal (Reflective) Material

- **Model:** Perfect or fuzzy reflection based on surface roughness
- **Implementation:** Reflect incident ray with optional random perturbation
- **Use Case:** Mirrors, polished metals, chrome

Reflection Visualization:



3.2.3 Emissive Material

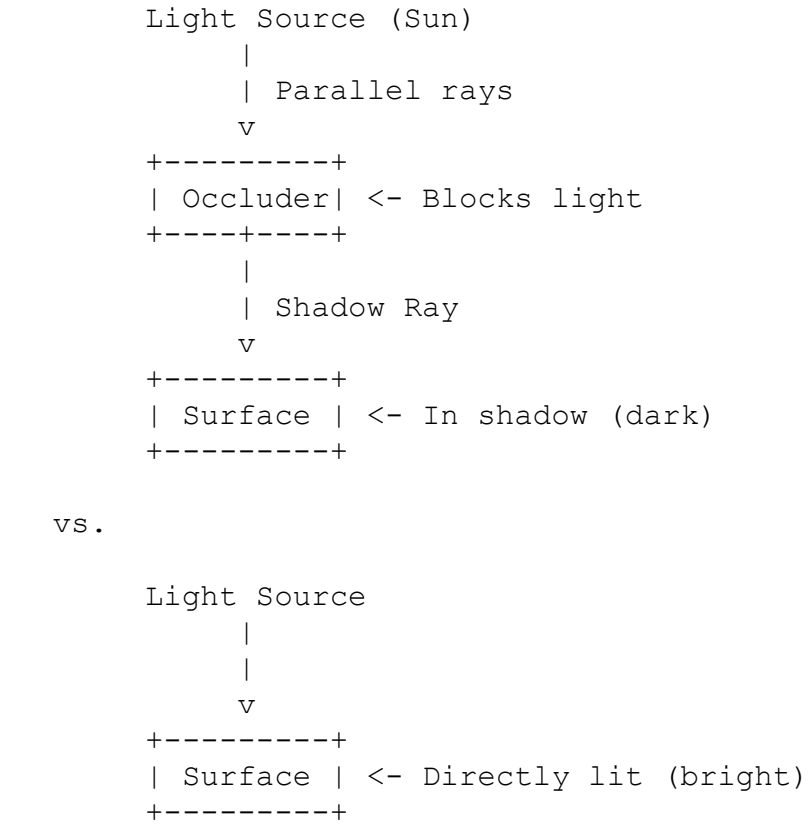
- **Model:** Light-emitting surfaces
- **Implementation:** Add emission color to ray color, no scattering
- **Use Case:** Light sources, glowing objects

3.3 Lighting and Shadows

3.3.1 Directional Light (Sun)

- **Model:** Parallel light rays from infinite distance
- **Shadow Testing:** Cast shadow ray from hit point to light source
- **Implementation:** Check for occlusions along shadow ray

Shadow Ray Visualization:



3.3.2 Soft Shadows

- **Model:** Area light sources create penumbra regions
- **Implementation:** Multiple shadow rays to different points on light source
- **Enhancement:** Monte Carlo sampling for realistic soft shadows

3.4 Anti-Aliasing

3.4.1 Multi-Sample Anti-Aliasing (MSAA)

- **Algorithm:** Cast multiple rays per pixel with random jitter
- **Sampling:** Stratified sampling or uniform random sampling
- **Quality:** Higher samples per pixel = less noise, smoother edges

---

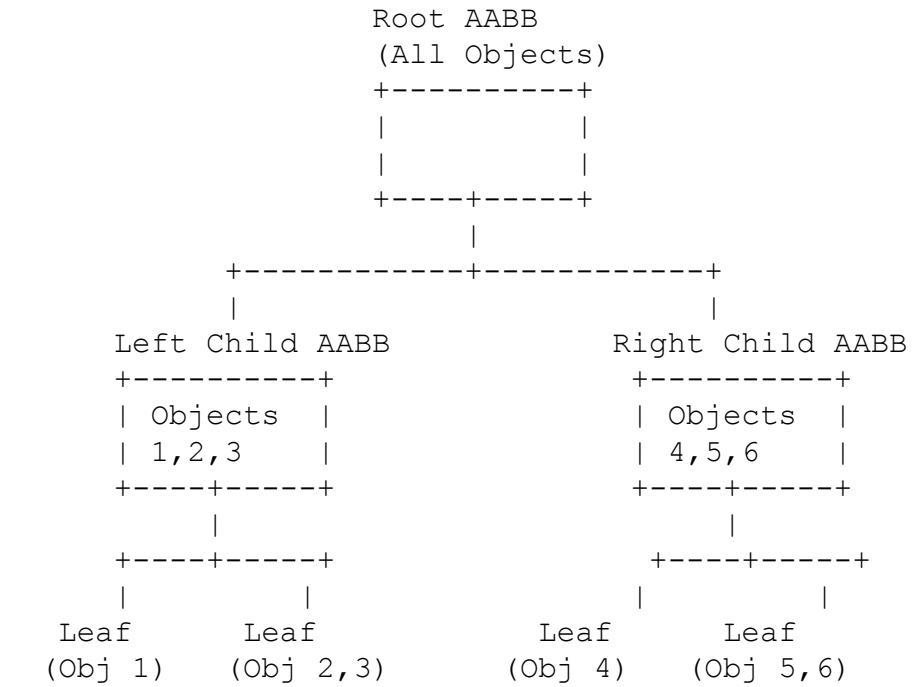
4. Optimization Techniques

4.1 Spatial Acceleration Structures

4.1.1 Bounding Volume Hierarchy (BVH)

- **Purpose:** Reduce ray-object intersection tests from O(n) to O(log n)
- **Implementation:** Recursive binary tree of axis-aligned bounding boxes
- **Construction:** Top-down approach using surface area heuristic (SAH)
- **Expected Speedup:** 10-100x for complex scenes

BVH Tree Structure:




```
Ray Traversal:
1. Test ray vs Root AABB
2. If hit, test children recursively
3. Only test objects in hit leaf nodes
```

4.1.2 KD-Tree (Alternative)

- **Purpose:** Alternative spatial partitioning structure
- **Trade-offs:** Better for static scenes, more complex construction
- **Use Case:** When BVH performance is insufficient

4.2 Parallel Processing

4.2.1 Multi-Threading

- **Approach:** Tile-based work distribution

- **Implementation:** Thread pool with work-stealing queue
- **Scalability:** Linear scaling up to CPU core count
- **Expected Speedup:** 4-16x on modern CPUs

Tile-Based Parallel Rendering:

```
Image (1920x1080) divided into tiles (64x64)

+-----+-----+-----+-----+
| T1  | T2  | T3  | T4  | <- Thread 1 processes
+-----+-----+-----+-----+
| T5  | T6  | T7  | T8  | <- Thread 2 processes
+-----+-----+-----+-----+
| T9  | T10 | T11 | T12 | <- Thread 3 processes
+-----+-----+-----+-----+
| T13 | T14 | T15 | T16 | <- Thread 4 processes
+-----+-----+-----+-----+

Work Distribution:
- Each thread picks next available tile
- Load balancing via atomic counter
- Cache-friendly: tiles processed sequentially
- Progress tracking per tile
```

4.2.2 SIMD Vectorization

- **Purpose:** Process multiple rays simultaneously
- **Implementation:** AVX/AVX2 intrinsics for 4-8 rays at once
- **Use Case:** Primary ray generation, vector math operations
- **Expected Speedup:** 2-4x for vectorizable code paths

4.3 Memory Optimization

4.3.1 Memory Pool Allocation

- **Purpose:** Reduce allocation overhead for temporary objects
- **Implementation:** Pre-allocated pools for rays, hit records
- **Benefit:** Lower memory fragmentation, better cache performance

4.3.2 Compact Data Structures

- **Purpose:** Improve cache efficiency
- **Implementation:** Structure-of-Arrays (SoA) for triangle data
- **Benefit:** Better vectorization, reduced cache misses

---

5. GPU Acceleration (Future Enhancement)

5.1 CUDA/OpenCL Implementation

- **Approach:** Parallelize ray tracing kernel on GPU
- **Challenges:** Memory management, recursion limits, dynamic data structures
- **Expected Speedup:** 10-100x for suitable scenes

5.2 GPU-Optimized Data Structures

- **BVH on GPU:** Compact representation, stackless traversal
- **Texture Memory:** Cache-friendly material and texture access
- **Shared Memory:** Tile-based rendering with shared ray batches

---

6. Tools and Technologies

6.1 Programming Language and Standards

- **C++20:** Modern features (concepts, ranges, coroutines potential)
- **Compiler:** GCC 10+ or Clang 12+ with full C++20 support
- **Rationale:** Performance-critical code, direct memory control, industry standard

6.2 Build System

- **CMake 3.16+:** Cross-platform build configuration
- **Presets:** Consistent build configurations across environments
- **Integration:** VS Code, CLion, and other IDE support

6.3 Third-Party Libraries

- **GLM (OpenGL Mathematics):** Vector and matrix operations
- **TinyXML2:** XML scene file parsing
- **OBJ_Loader:** Wavefront OBJ file format support
- **stb_image_write:** PNG/PPM image output
- **Rationale:** Well-tested, lightweight, header-only where possible

6.4 Development Tools

- **Version Control:** Git with GitHub for collaboration
- **CI/CD:** GitHub Actions for automated testing and builds
- **Debugging:** GDB/LLDB, AddressSanitizer for memory safety
- **Profiling:** Valgrind, perf, Intel VTune for performance analysis

---

7. Feature Development Layers

Layer 1: Functional Minimum

Goal: Basic working ray tracer that produces recognizable images

Features: - Basic ray generation from camera - Ray-sphere intersection - Simple Lambertian material (diffuse) - Single light source (directional sun) - Hard shadows - Basic anti-aliasing (1-4 samples per pixel) - PPM image output - XML scene file loading (camera, spheres, basic materials) - Command-line interface

Success Criteria: - Renders simple scene with 2-3 spheres - Produces image file successfully - Execution completes without crashes - Basic shadows visible

Estimated Effort: 40-50 hours

Layer 2: Minimum Goal

Goal: Reasonably complete ray tracer with essential features

Features: - All Layer 1 features - Ray-triangle intersection - OBJ mesh file loading - Multiple material types (Lambertian, Metal, Emissive) - MTL material file support - Recursive ray tracing (reflections) - Configurable ray depth (3-10 bounces) - Improved anti-aliasing (10-20 samples per pixel) - PNG image output with gamma correction - Multi-threaded rendering (tile-based) - Progress reporting - Basic error handling and validation

Success Criteria: - Renders complex scenes with meshes - Realistic reflections on metal surfaces - Smooth anti-aliasing - Acceptable render times (< 5 minutes for 800x600) - Professional image quality

Estimated Effort: 60-80 hours

Layer 3: Desired Goal

Goal: High-quality ray tracer with advanced features and optimizations

Features: - All Layer 2 features - Bounding Volume Hierarchy (BVH) acceleration structure - Soft shadows (area lights) - Advanced materials (dielectric/glass with refraction) - Texture mapping support - Monte Carlo path tracing - Importance sampling - Tone mapping and HDR output - Configurable rendering parameters via CLI - Performance profiling and benchmarking tools - Comprehensive scene validation - Memory optimization (pool allocation)

Success Criteria: - Renders photorealistic images - Handles complex scenes (1000+ triangles) efficiently - Render times < 2 minutes for 1920x1080 - Professional-quality output comparable to reference renderers - Well-documented codebase

Estimated Effort: 80-100 hours

Layer 4: Maximum Goal

Goal: Production-quality ray tracer with cutting-edge features

Features: - All Layer 3 features - KD-Tree as alternative acceleration structure - Advanced materials (subsurface scattering, anisotropic) - Environment maps (HDRI skyboxes) - Depth of field (camera aperture simulation) - Motion blur - Volumetric rendering (fog, participating media) - Denoising algorithms (bilateral filter, ML-based) - SIMD optimizations (AVX/AVX2) - Interactive preview mode (progressive rendering) -

Scene export/import formats (glTF, PBR materials) - Comprehensive test suite - Performance comparison with reference implementations

Success Criteria: - Industry-grade image quality - Handles very complex scenes (10,000+ triangles) - Real-time preview at low resolution - Comprehensive documentation and examples - Publication-ready results

Estimated Effort: 100-150 hours

Layer 5: Extras

Goal: Research-level extensions and experimental features

Features: - GPU acceleration (CUDA or OpenCL) - Real-time ray tracing capabilities - Advanced global illumination (photon mapping, radiosity) - Bidirectional path tracing - Metropolis Light Transport (MLT) - Spectral rendering (wavelength-dependent) - Participating media with multiple scattering - Advanced denoising (OptiX AI, Intel Open Image Denoise) - Distributed rendering (network parallelization) - Web-based interface or GUI - Integration with Blender/Maya as plugin - Scientific paper documenting algorithms and results

Success Criteria: - Real-time or near-real-time rendering - Research contributions to rendering algorithms - Potential for academic publication - Industry adoption or open-source community interest

Estimated Effort: 200+ hours

---

8. Task List and Schedule

8.1 Task Breakdown

Task ID	Task Description	Who	Estimated Hours	Actual Hours
1	Project setup and architecture design	Student	8	-
2	Core math library (vec3, ray, utilities)	Student	12	-
3	Ray-sphere intersection implementation	Student	6	-
4	Basic camera system	Student	8	-
5	Lambertian material implementation	Student	6	-
6	Basic ray tracing loop	Student	10	-
7	XML scene file parser	Student	12	-
8	Image output (PPM format)	Student	4	-
9	Testing and debugging basic renderer	Student	8	-
10	Ray-triangle intersection	Student	8	-
11	OBJ mesh file loader	Student	12	-
12	Metal material with reflections	Student	8	-
13	Emissive material	Student	4	-
14	Recursive ray tracing	Student	10	-
15	Shadow ray implementation	Student	6	-
16	Multi-sample anti-aliasing	Student	8	-
17	PNG output with gamma correction	Student	6	-
18	MTL material file parser	Student	10	-

Task ID	Task Description	Who	Estimated Hours	Actual Hours
19	Multi-threaded rendering (tile-based)	Student	16	-
20	Progress reporting and CLI improvements	Student	6	-
21	BVH acceleration structure	Student	24	-
22	Soft shadows implementation	Student	12	-
23	Dielectric/glass material	Student	12	-
24	Texture mapping support	Student	16	-
25	Monte Carlo path tracing	Student	20	-
26	Tone mapping and HDR	Student	8	-
27	Performance profiling and optimization	Student	16	-
28	KD-Tree acceleration structure	Student	20	-
29	Environment maps (HDRI)	Student	12	-
30	Depth of field	Student	10	-
31	Motion blur	Student	12	-
32	SIMD optimizations	Student	16	-
33	Denoising algorithms	Student	20	-
34	GPU acceleration research and planning	Student	12	-
35	CUDA/OpenCL implementation	Student	40	-
36	Documentation and report writing	Student	20	-
37	Testing and validation	Student	16	-
38	Presentation preparation	Student	8	-

Total Estimated Hours (Layers 1-3): ~350 hours
Total Estimated Hours (Layer 4): ~500 hours
Total Estimated Hours (Layer 5): ~700+ hours

8.2 Milestones and Deliverables

Milestone 1: Basic Ray Tracer (Week 4)

- **Deliverables:**
  - Working ray-sphere intersection
  - Simple scene rendering
  - Basic image output
- **Tasks:** 1-9
- **Success Criteria:** Renders 2-3 spheres with basic lighting

Milestone 2: Complete Core Engine (Week 8)

- **Deliverables:**
  - Full material system
  - Mesh loading
  - Recursive reflections
  - Multi-threading
- **Tasks:** 10-20
- **Success Criteria:** Renders complex scenes with meshes, reflections, shadows

Milestone 3: Optimized Renderer (Week 12)

- **Deliverables:**
  - BVH acceleration
  - Advanced materials
  - Path tracing
  - Performance optimizations
- **Tasks:** 21-27
- **Success Criteria:** Fast rendering of complex scenes, photorealistic quality

Milestone 4: Advanced Features (Week 16)

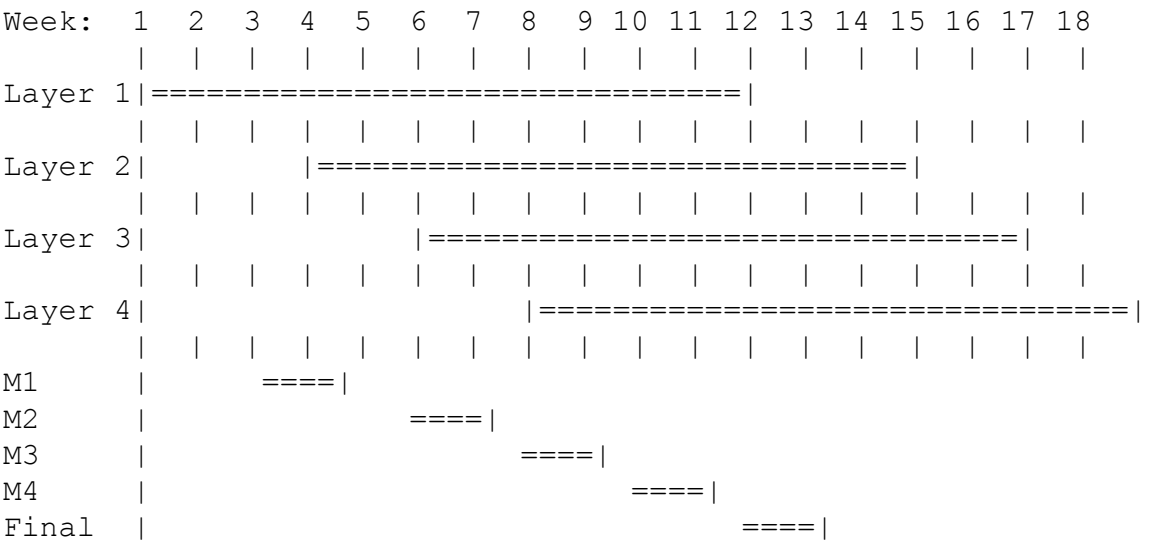
- **Deliverables:**
  - All Layer 3 features complete
  - Advanced rendering effects
  - Comprehensive documentation
- **Tasks:** 28-33
- **Success Criteria:** Production-quality renderer

Milestone 5: Final Submission (Week 18)

- **Deliverables:**
  - Complete project
  - Final report
  - Presentation
  - Demo videos
- **Tasks:** 34-38
- **Success Criteria:** All deliverables submitted, presentation completed

8.3 Detailed Schedule

Gantt Chart Visualization:



Weeks 1-4: Foundation (Layer 1)

- **Week 1:** Project setup, architecture design, core math library
- **Week 2:** Basic ray tracing, sphere intersection, camera
- **Week 3:** Materials, lighting, XML parser
- **Week 4:** Integration, testing, Milestone 1

Weeks 5-8: Core Features (Layer 2)

- **Week 5:** Triangle intersection, mesh loading
- **Week 6:** Advanced materials, recursive tracing
- **Week 7:** Multi-threading, anti-aliasing
- **Week 8:** Polish, testing, Milestone 2

Weeks 9-12: Optimization (Layer 3)

- **Week 9:** BVH implementation
- **Week 10:** Path tracing, advanced materials
- **Week 11:** Performance optimization
- **Week 12:** Testing, Milestone 3

Weeks 13-16: Advanced Features (Layer 4)

- **Week 13:** Advanced effects (DOF, motion blur)
- **Week 14:** SIMD, denoising
- **Week 15:** Final optimizations
- **Week 16:** Milestone 4, documentation

Weeks 17-18: Finalization

- **Week 17:** GPU research (if time permits)
- **Week 18:** Report writing, presentation prep, final submission

---

9. Risk Assessment and Mitigation

9.1 Technical Risks

Risk	Impact	Probability	Mitigation
Performance not meeting targets	High	Medium	Early profiling, optimize critical paths, use acceleration structures
Complex bugs in ray tracing	High	Medium	Comprehensive testing, reference implementations, incremental development
Memory issues with large scenes	Medium	Low	Memory profiling, efficient data structures, streaming for large meshes

Risk	Impact	Probability	Mitigation
GPU implementation complexity	Medium	High	Start with CPU, GPU as optional enhancement, use proven frameworks

9.2 Schedule Risks

Risk	Impact	Probability	Mitigation
Underestimating task complexity	High	Medium	Buffer time in schedule, prioritize core features
Scope creep	Medium	High	Strict layer-based development, defer extras to Layer 5
External dependencies issues	Low	Low	Use stable libraries, have alternatives ready

---

10. Success Metrics

10.1 Functional Metrics

- **Target:** Render scenes with 1000+ triangles (Layer 3)
- **Target:** Support all planned material types (Lambertian, Metal, Emissive, Dielectric)
- **Target:** Produce photorealistic images comparable to industry standards
- **Target:** Handle complex lighting scenarios including shadows, reflections, and refractions

10.2 Performance Metrics

- **Target:** Render 1920x1080 scene in < 2 minutes (Layer 3)
- **Target:** 10-50x speedup with BVH (vs. naive)
- **Target:** Linear scaling with thread count (up to CPU cores)

10.3 Quality Metrics

- **Image Quality:** Comparable to reference renderers (POV-Ray, Mitsuba)
- **Code Quality:** Well-documented, modular, extensible
- **Robustness:** Handles edge cases, validates input, graceful error handling

---

11. Conclusion

This project specification outlines the development of a comprehensive 3D ray tracing engine that will serve as both a learning exercise and a demonstration of advanced computer graphics techniques. The layered

approach ensures a functional system at each stage while allowing for ambitious enhancements as time permits.

The focus on fundamental algorithms, optimization techniques, and modern C++ practices will provide valuable experience in computer graphics programming. The modular design ensures the codebase remains maintainable and extensible, allowing for future enhancements and research contributions.

By following this specification, the project will deliver a production-quality ray tracing engine capable of generating photorealistic images while demonstrating deep understanding of computer graphics principles, software engineering best practices, and performance optimization techniques.

Appendix A: References and Resources

Academic Papers

- Whitted, T. (1980). "An improved illumination model for shaded display"
- Kajiya, J. T. (1986). "The rendering equation"
- Cook, R. L., et al. (1984). "Distributed ray tracing"
- Veach, E. (1997). "Robust Monte Carlo methods for light transport simulation"

Books

- Shirley, P., & Morley, R. K. (2003). "Realistic Ray Tracing"
- Pharr, M., Jakob, W., & Humphreys, G. (2016). "Physically Based Rendering: From Theory to Implementation"

Online Resources

- Ray Tracing in One Weekend series (Peter Shirley)
- Scratchapixel.com - Computer Graphics Programming
- NVIDIA OptiX documentation
- Intel Embree documentation

Document Version: 1.0
Last Updated: [Date]
Status: Proposal