A logo of a roman soldier

Description automatically generated

**CS3722 – Formal languages &**

**The theory of Computation**

**Optimization Algorithm for Ray Tracing –**

**Computer Graphics**

|  |  |  |  |
| --- | --- | --- | --- |
| **Group 6** | | | |
| **Group Members** | **No.** | **Name** | **ID** |
| 1 | Nguyen Quang Trung | 1624786 |
| 2 | Le Duc Tuyen | 1624790 |
| 3 | Nguyen Viet Anh Tuan | 1624788 |
| 4 | Phung Minh Quang | 1624752 |
| **Supervior** | Dr. Dinh Han Nguyen | | |

*- Ha Noi, August 21st, 2025 -*

**Preface**

This report, Optimization Algorithm for Ray Tracing – Computer Graphics, was prepared as part of the requirements for the course CS372 – Formal Languages under the guidance of Dr. Dinh Han Nguyen. The purpose of this project is to study and implement a CPU-based ray tracing renderer, with particular focus on optimization strategies such as multithreading and Bounding Volume Hierarchies (BVH).

Through the project, our group aimed not only to build a functioning ray tracer but also to explore the balance between educational clarity and computational efficiency. The report reflects our learning process, the challenges we encountered, and the solutions we designed to achieve meaningful results.

We would like to express our gratitude to Dr. Nguyen for his valuable instruction and support throughout the course. We also appreciate the teamwork and dedication of all group members, whose combined efforts made the completion of this project possible.

We hope this report provides useful insights into the fundamentals of ray tracing and its optimization, serving as both a record of our work and a reference for future study in computer graphics.

Table of Contents

[1 General Description of Project 3](#_Toc206623769)

[1.1 Introduction to Ray Tracing 3](#_Toc206623770)

[1.2 Objectives 3](#_Toc206623771)

[1.3 Concerning Problems 4](#_Toc206623772)

[1.3.1 Intersection Testing Complexity 4](#_Toc206623773)

[1.3.2 Multiple Rays Per Pixel 4](#_Toc206623774)

[1.3.3 Recursive Ray Paths 4](#_Toc206623775)

[1.3.4 Data Structures and Memory Access 5](#_Toc206623776)

[1.3.5 Sequential Nature of the CPU 5](#_Toc206623777)

[1.4 Team Members Contribution 5](#_Toc206623778)

[2 Software Requirements Specifications 6](#_Toc206623779)

[2.1 Functional requirements 6](#_Toc206623780)

[2.1.1 Correct Ray Simulation 6](#_Toc206623781)

[2.1.2 Scene Rendering 6](#_Toc206623782)

[2.1.3 Optimization Features 6](#_Toc206623783)

[2.2 Non-Functional Requirements 6](#_Toc206623784)

[2.2.1 Simplicity and Transparency 6](#_Toc206623785)

[2.2.2 Performance Constraints 6](#_Toc206623786)

[2.2.3 Correctness Over Speed 7](#_Toc206623787)

[3 Proposed Solution and Strategy 8](#_Toc206623788)

[4 Methodology 9](#_Toc206623789)

[4.1 Rendering Technique 9](#_Toc206623790)

[4.2 Mathematical Tools 10](#_Toc206623791)

[4.3 Acceleration Structure 11](#_Toc206623792)

[4.4 Parallelization Strategy 11](#_Toc206623793)

[5 Tools and Development Environment 12](#_Toc206623794)

[6 Experimental Setup and Testing 13](#_Toc206623795)

[6.1 Scene setup 13](#_Toc206623796)

[6.2 Configurations Breakdown 14](#_Toc206623797)

[6.3 Optimization Algorithm – Bounding Volume Hierarchy 15](#_Toc206623798)

[6.4 Performance comparison 18](#_Toc206623799)

[7 Conclusion and further discussion 19](#_Toc206623800)

[8 Appendix 20](#_Toc206623801)

# General Description of Project

## Introduction to Ray Tracing

Ray tracing is a computer graphics rendering technique that simulates the physical behavior of light in order to generate photorealistic images.

By tracing the paths of virtual light rays as they interact with objects in a 3D scene (through reflection, refraction, scattering, and shadowing), ray tracing can naturally reproduce effects like soft shadows, mirror reflections, transparency, and global illumination.

Although rasterization remains the industry standard for real-time applications such as games, ray tracing is of great academic interest due to its elegant physical formulation and ability to approximate real-world optics. Studying ray tracing offers invaluable educational insights into geometry, linear algebra, physics, and algorithm design, making it a powerful tool for learning the fundamentals of computer graphics.



## Objectives

The main objective of the project is to:

* Educational Clarity: To design and implement a ray tracing renderer in C++ that clearly demonstrates the mathematical and algorithmic foundations of ray tracing, including ray–object intersections, shading, and recursion.
* Optimization Techniques: To investigate and apply lightweight optimization strategies, such as multithreading and Bounding Volume Hierarchies (BVH), to reduce computational cost while maintaining algorithmic transparency.
* Performance Evaluation: To analyze the trade-off between image quality and rendering time by experimenting with different parameters (resolution, samples per pixel, recursion depth) and comparing results with and without optimization.
* Practical Implementation: To gain hands-on experience with C++ programming, parallel processing, and data structures in the context of computer graphics.
* Knowledge Contribution: To provide a structured, educational project report that documents the implementation process, discusses challenges, and highlights future directions such as GPU acceleration and advanced shading models.

## Concerning Problems

### Intersection Testing Complexity

In a typical ray-traced scene, there may be dozens, hundreds, or even thousands of objects. A naive ray tracer checks each ray against every object, to find the nearest hit:

• If the image is 800×600 and 100 rays are shot per pixel → 48 million rays

• With 200 scene objects → 9.6 billion intersection tests

### Multiple Rays Per Pixel

A grid with red dots and black text

AI-generated content may be incorrect.

High-quality rendering relies on shooting many rays per pixel to reduce noise (Monte Carlo sampling):

• Anti-Aliasing → jittered primary rays

• Soft Shadows → shadow rays towards extended light sources

• Glossy Reflections / Global Illumination → randomly scattered secondary rays

Every extra ray multiplies the number of intersection tests — so quality vs speed becomes a serious trade-off.

### Recursive Ray Paths

Ray tracing is inherently recursive:

• A primary ray hits an object → spawn reflections, refractions, shadow rays, and sometimes GI rays.

• Each of those rays may recursively produce more rays.

• Even with shallow recursion depths (3–5 bounces), the number of rays grows exponentially.

CPUs struggle with this branching behavior, especially since ray paths diverge significantly, leading to poor cache usage and unpredictable execution patterns.

### Data Structures and Memory Access

Efficient traversal requires spatial acceleration structures (like BVH), but:

• Building these structures takes time and complicates the code.

• Traversing them involves pointer chasing and branching, which CPUs handle slower compared to GPUs.

• If written poorly, the acceleration structure itself becomes a bottleneck.

### Sequential Nature of the CPU

While a GPU might trace thousands of rays in parallel, a CPU traditionally processes tasks sequentially or with a small number of cores:

• Without multithreading, rendering even a modest scene becomes extremely slow.

• Even with multithreading, CPUs cannot match the massive parallelism of GPUs, so careful workload balancing and synchronization are needed to see performance gains.

## Team Members Contribution

|  |  |  |
| --- | --- | --- |
| **No.** | **Full Name** | **Role** |
| 1 | Nguyen Quang Trung | Leader, core features, BVH algorithm implementation |
| 2 | Le Duc Tuyen | Core features, C++ multithreading implementation. |
| 3 | Nguyen Viet Anh Tuan | Core features, SDL2 Library implementation |
| 4 | Phung Minh Quang | Additional features, code clean-up, testing, finalization |

# Software Requirements Specifications

The core challenge lies in **balancing educational simplicity** (so that users can follow the math and logic), with **enough optimization** to avoid unbearably slow render times — all while constrained to CPU execution. This tension between *clarity* and *efficiency* fundamentally defines the research problem in our project.

## Functional requirements

### Correct Ray Simulation

The renderer must accurately simulate ray behavior and produce meaningful visual output.

* The output images should demonstrate:
* Hard and soft shadows
* Specular reflections (mirror-like surfaces)
* Diffuse shading (Lambertian surfaces)
* Recursive ray interactions

### Scene Rendering

Even simple test scenes (e.g., spheres on a plane) must resemble their real-world lighting counterparts. The renderer must allow parameter adjustments (samples per pixel, recursion depth, resolution) to influence the quality of results.

### Optimization Features

The system must support basic optimizations to improve performance without altering correctness, specifically:

* Multithreading for parallel pixel/ray processing.
* Bounding Volume Hierarchy (BVH) for efficient intersection checks.

## Non-Functional Requirements

### Simplicity and Transparency

The project emphasizes readability and clarity rather than raw performance. The source code is designed to explicitly demonstrate the mathematics behind ray tracing, such as ray–object intersections, vector algebra, shading equations, and recursion. To support learning, architecture follows a modular and intuitive structure, with separate classes for components like rays, spheres, cameras, materials, and BVH. Unnecessary abstractions and heavy external libraries are avoided so that the logic remains visible and easily editable, even for beginners.

### Performance Constraints

Although the renderer runs entirely on the CPU, it must still achieve reasonable performance. The goal is to render small to medium test scenes (10–200 objects, resolutions of 400×400 to 800×800, and 10–50 samples per pixel) within a tolerable timeframe of seconds to minutes. At the same time, performance improvements must be carefully balanced with clarity: the system should be fast enough for experimentation, yet simple enough that learners can still follow the underlying algorithm.

### Correctness Over Speed

In this project, accuracy takes priority over speed or photorealism. The renderer should consistently produce physically correct results, even if this requires longer rendering times. A slower but correct output is considered preferable to a faster result that compromises correctness.

# Proposed Solution and Strategy

To satisfy these requirements, we implement a CPU-based ray tracer written in C++ and deliberately designed for educational clarity. The program uses a recursive ray casting approach to simulate light interactions and generate images. Although executing entirely on the CPU, the following strategies are used to improve performance without sacrificing transparency:

• Multithreading — the renderer is parallelized using standard C++ threads, allowing multiple rays to be processed simultaneously across all available CPU cores.

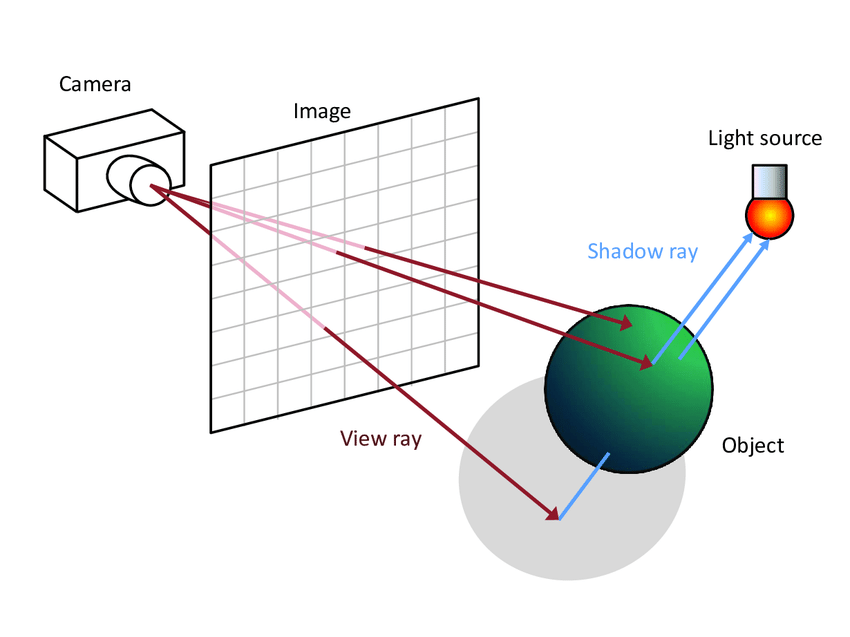
• BVH (Bounding Volume Hierarchy): a simple spatial acceleration structure is used to reduce the number of ray-object intersection tests. By hierarchically grouping objects into bounding boxes, the ray tracer can quickly eliminate large portions of the scene during traversal.

• Simplified materials and integrator — shading models and light transport are kept intentionally simple to highlight essential ray tracing principles rather than advanced physically based rendering concepts.

A GPU-based implementation (e.g., using Vulkan or DirectX Ray Tracing) is acknowledged as a natural future direction to achieve real-time performance. However, it is excluded in this project due to its significantly higher technical barrier and the desire to maintain algorithmic visibility appropriate for educational purposes.

# Methodology

## Rendering Technique



The renderer is based on Whitted-style ray tracing, a classic recursive algorithm introduced by Turner Whitted in 1980. For each pixel on the image plane, we have a virtual camera shooting primary ray into its position, eventually make it to the scene. If the ray hits an object, secondary rays are recursively spawned to simulate:

• Reflection rays (mirror-like bounces),

• Refraction rays (transparent materials),

• Shadow rays (testing visibility toward light sources).

This recursive strategy allows the renderer to produce effects such as reflections, shadows, and simple transparency without relying on approximations used in rasterization.

## Mathematical Tools

A diagram of a graphing of a graph

AI-generated content may be incorrect.

To support accurate ray–object interaction, the implementation relies on:

• Vector algebra — for computations involving directions, positions, and normals in 3D space.

• Intersection algorithms — such as ray–sphere intersection tests, which require solving quadratic equations.

• Shading models — including Lambertian (diffuse) reflectance and perfect specular reflection, to convert geometric intersections into pixel color contributions.

• Recursive color accumulation — where the final pixel color is determined by repeatedly combining contributions from direct lighting and reflected/refracted rays along a ray path.

## Acceleration Structure



To avoid the inefficiency of testing every ray against every object, a Bounding Volume Hierarchy (BVH) is constructed. Objects in the scene are grouped into a binary tree of axis-aligned bounding boxes (AABBs). During rendering, rays first test for intersection with these boxes:

• If a ray misses a box, all objects inside are skipped.

• If it hits, the algorithm recursively traverses the box’s children.

This dramatically reduces the number of expensive intersection tests, especially in scenes with many objects.

This helps us achieve a complexity level of O (log n) instead of O(n) with naïve ray

## Parallelization Strategy

Even on the CPU, rendering can be parallelized by distributing pixels or scanlines across threads. The program uses multithreading via the C++ Standard Library (std::thread) to spawn multiple worker threads:

• Each thread is responsible for ray tracing a subset of the image.

• Color results are written to a shared framebuffer.

This enables full utilization of modern multi-core CPUs, improving performance without changing the educational nature of the code.

# Tools and Development Environment

Programming Language: C++ — chosen for its performance, low-level memory control, and alignment with real-world graphics engines.

IDE: Visual Studio — provides convenient debugging and multithreading support on Windows platforms.

Libraries:

• STL (<thread> and <mutex>) — for lightweight multithreaded execution.

• SDL2 — for handling window creation and displaying the rendered image to the screen.

• chrono: for calculating the time required to finish rendering an output.

• vector: dynamic arrays for keeping the records of the 3d objects

• iostream: for printing debug or progress information to the console

# Experimental Setup and Testing

## Scene setup

A math equations on a white background

AI-generated content may be incorrect.

First, we need to set up some rendering parameters:

• **aspect\_ratio**: Screen aspect ratio for width / height.

• **image\_width**: image resolution width, divided by aspect ratio to calculate image\_height.

• **samples\_per\_pixel**: the amount of wanted rays per pixel, increasing it means better image quality, but worse performance.

• **max\_depth**: the wanted amount of bounce for each light ray, increasing it helps to provide more correct image output, but also slows down rendering.

• **background**: the color assigned for pixels not hit by the rays, we assign the RGB value of (0.7, 0.8, 1.0) to represent a simple blue sky.

Now, we render a simple scene with around 40 spheres, a big sphere underneath to represent the ground, and where the rays don’t hit anything, we assign color blue to represent the sky.

A group of colorful balls

AI-generated content may be incorrect.

This image was rendered with 1000 samples per pixel, **max\_depth** of 10 and resolution of 1920 x 1080. Now let’s try to scale it down to breakdown possibilities when altering the metrics.

## Configurations Breakdown

First, let’s try to render the same scene with much lower resolution of 800x450 and just 1 sample per pixel, with **max\_depth** of 2 to see the result and the required time to render:

A group of balls on a green surface

AI-generated content may be incorrect.

The result is a very grainy, noisy image, and the reflections, how light is scattered are all incorrect. It took 5.675 seconds to render the image.

Now we increase **max\_depth** to 5 to improve the reflections and light scattering:

A group of colorful balls on a green surface

AI-generated content may be incorrect.

The result now is much different compared to the previous, now we can see proper reflections of the left sphere, which is assigned with “metallic” material, and the middle sphere with “dielectric” material to simulate the reflections of water or special types of glass. The result took 7.456 seconds to render, which is 30% slower. However, the image is still quite noisy and grainy, which we now need to increase **samples\_per\_pixel** to 20 and see the result:

A group of colorful balls

AI-generated content may be incorrect.

The image now appears much finer and crisper, although to achieve a better result in large scale productions, **samples\_per\_pixel** can be up to thousands. However, this leads us to a big problem, the scene now takes 121.842 seconds, which is 21x slower than the first image, which shouldn’t be acceptable for such a simple scene.

## Optimization Algorithm – Bounding Volume Hierarchy

In the naïve ray tracing technique, every ray have to loop through every object in the scene to check for intersection:

A screen shot of a computer code

AI-generated content may be incorrect.

We have a simple for each loop to loop through objects, and we have the function hit() to detect intersection. This is too performance costly when the hit() function takes O(n) complexity for every ray, which significantly increases rendering time.

To solve this problem, we need a more efficient optimization algorithm. In our implementation, we used a simple algorithm called Bounding Volume Hierarchy.

Firstly, we can simply define objects coordinates and wrap them inside virtual 3d boxes called “Bounding Box”:

A black background with orange figures

AI-generated content may be incorrect.

A Bounding Box is a simple structure describing an axis-aligned bounding box, or AABB. It contains two points, one identifying the minimum point on the box, and the other identifying the maximum point. The minimum point is where the x, y, and z coordinates are all smallest, and the maximum point is where those coordinates are all largest.

Every box can be grouped with another box, and those 2 boxes can be left and right nodes of a parent box, and they are all components of a hierarchical tree like this:



In our program, we used this function **bvh\_node**() to define bounding boxes within the scene. This function constructs a **BVH (Bounding Volume Hierarchy) node** for a set of objects:

* **Compute the bounding box** that encloses all objects in the range [start, end).
* **Choose a split axis** (x, y, or z) based on which dimension of the bounding box is longest.
* **Sort the objects** along that axis to group them spatially.
* **Split the objects** into two halves:
  + If there’s only 1 object → both left and right children point to it.
  + If there are 2 objects → assign one to left, one to right.
  + Otherwise → recursively create left and right BVH nodes from the two halves.

A screenshot of a computer code

AI-generated content may be incorrect.

We then have a different hit() function to define if rays hit the bounding boxes, if a ray misses a parent box, then we can skip checking for all the child boxes within it:

A screen shot of a computer code

AI-generated content may be incorrect.

The result is a binary tree where each node contains a bounding box, allowing rays to quickly skip groups of objects during rendering. With that method, we can simply achieve the level of O (log n) complexity for each ray test, which significantly reduces rendering time. Now let’s see the result of the previous scene with 20 **samples\_per\_pixel** and **max\_depth** of 5 that took us 121.842 seconds to render:

A group of colorful balls

AI-generated content may be incorrect.

Now this scene took us around 9 seconds to render, which is unbelievably faster than without BVH algorithm enabled.

## Performance comparison

The project successfully implemented a CPU-based ray tracer with key optimizations in multithreading and Bounding Volume Hierarchy (BVH). Experimental results demonstrated that these techniques significantly improve rendering performance while preserving correctness and educational clarity.

The following table summarizes our main findings:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Configuration** | **Resolution** | **Samples/Pixel** | **Max Depth** | **Render Time (s)** | **Notes** |
| Baseline (no BVH) | 800×450 | 1 | 2 | 5.675 | Grainy, incorrect light scattering |
| Higher depth, no BVH | 800×450 | 1 | 5 | 7.456 | Better reflections, still noisy |
| Higher quality, no BVH | 800×450 | 20 | 5 | 121.842 | Much clearer, but very slow |
| With BVH | 800×450 | 20 | 5 | ~9.0 | Similar quality, dramatically faster |

# Conclusion and Further Discussion

Our program runs on Visual Studio, which by default only uses the power of single core CPU, so all rays are traced sequentially and that limits performance. To take advantage of full CPU performance, we used the feature **multithreading** of C++ from the very start to significantly boost rendering time by splitting the image into chunks that can be processed simultaneously across multiple threads.

Our program was implemented on a personal laptop that has the **CPU AMD Ryzen 5 5600h** with 6 cores and 12 threads. Using **multithreading** allows us to use up to 12 threads to run in parallel, roughly corresponding to a theoretical 12x speedup compared to single-threaded execution. In practice, the actual speedup is slightly lower due to overhead from thread management and the fact that some core might remain idle during short tasks, but we still observed around 8-10x faster rendering for most scenarios.

Multithreading helps especially for CPU-based ray tracing because each pixel (or block of pixels) can be computed independently — there are no dependencies between rays. This makes the workload highly parallelizable and allows the CPU’s multiple cores to be fully utilized, significantly reducing the total render time.

From the results gathered, we conclude that BVH reduces intersection complexity from O(n) to O(log n) and that multithreading allows near-linear speedup on multi-core CPUs. Together, these optimizations make real-time experimentation feasible while maintaining simplicity and transparency for educational purposes.

# Appendix

This project has been completed in the span of 6 weeks according to the requirements of the course.

The timeline of this project is as follows:

• Week 1: Initial research and familiarization with the core concepts of ray tracing and Bounding Volume Hierarchies (BVH). Team members will be assigned specific tasks, and project milestones will be defined.

• Week 2: Focus on the development of the core components, including BVH construction, ray traversal implementation, and preliminary performance testing.

• Week 3: Integration of individual components, performance evaluation, and optimization. Team members will also work on visualizations and refining results.

• Week 4-6: Final testing, analysis, and preparation of the report and presentation materials, including recommendations for future improvements.