Path Tracing Renderer Using Monte Carlo Methods

Silas Maughan

October 24, 2024

Abstract

This report presents an expository on the foundational mathematical knowledge and implementation of a path tracing renderer using Monte Carlo methods to simulate realistic lighting in a 3D scene. Various sampling techniques and variance reduction methods are explored to enhance image quality and convergence speed. Experimental results demonstrate the effectiveness of these techniques in reducing noise and improving rendering efficiency.

Contents

1 Introduction

1.1 Purpose and Scope

1.1.1 Goals of the Document

This expository aims to achieve the following objectives:

1. To provide a comprehensive analysis of the *mathematical* foundations underlying path tracing, creating a scene, and Monte Carlo integration techniques.

2. To demonstrate the *visual* impact of Monte Carlo methods in path tracing for realistic light simulation, exploring their impact on image fidelity, convergence rates, and rendering efficiency.

3. To elucidate the *design* and implementation of a path tracing system, with particular emphasis on acceleration structures like Bounding Volume Hierarchies (BVH), real-time visualisation with OpenGL and adhering to software modelling principles.

Through these objectives, this document seeks to offer a thorough exploration of path tracing, from its mathematical underpinnings to its practical realization in computer graphics, serving as a valuable resource for researchers and practitioners in the field of physically-based rendering.

1.1.2 Overview of the Methodological Approach

This expository adopts a progressive approach to the principles and implementation of path tracing. We begin by examining the foundational elements of a generic path tracer, including ray representation, intersection testing, and basic shading models. For each of these components, we present the underlying mathematical principles, followed by their corresponding code implementations. This foundational phase establishes the core concepts upon which more advanced techniques are built.

Building upon these foundational elements, we then abstract the entire lighting simulation process into a comprehensive mathematical framework. This transition allows us to view the path tracing problem as an integration task. We introduce Monte Carlo methods as a powerful tool for numerical integration, exploring how these statistical techniques can be applied to solve the rendering equation efficiently.

As we introduce each component, we demonstrate its effect on the rendered image, allowing for a clear understanding of how theoretical concepts translate to visual outcomes. This iterative approach provides immediate feedback on the impact of each implemented feature, reinforcing the connection between mathematical principles and their applications in computer graphics.

1.2 Background

1.2.1 Overview of Ray Tracing

Ray tracing is a rendering technique that simulates the physical behavior of light to create realistic images. At its core, ray tracing involves tracing the path of light rays as they interact with objects in a virtual scene [2]. The fundamental concept is to cast rays from a virtual camera through each pixel of an image plane into the scene. These rays interact with objects, potentially reflecting, refracting, or being absorbed, mimicking the behavior of light in the real world. By accurately modeling these interactions, ray tracing can produce highly realistic effects such as shadows, reflections, and refractions.

Path tracing, an advanced form of ray tracing, extends this concept by using Monte Carlo methods to solve the rendering equation [4]. It traces numerous light paths through the scene, accounting for multiple bounces and complex light interactions. This approach allows for the accurate simulation of global illumination effects, including soft shadows, color bleeding, and caustics [8].

The power of ray tracing and path tracing lies in their ability to naturally handle a wide range of lighting phenomena, producing physically accurate images. However, this accuracy comes at the cost of increased computational complexity, often requiring sophisticated optimization techniques to achieve reasonable rendering times [6].

1.2.2 Importance in Computer Graphics

Path tracing has become a cornerstone technique in modern computer graphics, particularly in applications demanding high levels of photorealism. Its ability to accurately simulate complex light interactions makes it invaluable in industries such as film and television visual effects, architectural visualization, product design, and video game development [5].

In the film industry, path tracing is used to create photorealistic CGI that seamlessly blends with live-action footage [1]. Moreover, with the advent of real-time ray tracing in consumer hardware, path tracing techniques are increasingly being adopted in interactive applications like video games, pushing the boundaries of real-time graphics fidelity [7].

2 Ray Tracing Fundamentals

2.1 Rays

2.1.1 A Ray of Light as a Vector

In physics, light is an electromagnetic wave that propagates through space. However, in many scenarios, particularly in computer graphics, we can approximate light behavior using the concept of rays. This simplification, known as geometric optics, is valid when the wavelength of light is much smaller than the objects it interacts with.

A ray of light can be thought of as an idealized narrow beam of light traveling in a straight line. This approximation allows us to model light propagation without dealing with the complexities of wave optics, making it computationally feasible for rendering purposes.

In mathematics, a vector is a quantity that has both magnitude and direction. It can be represented as an arrow in space, defined by its starting point and its direction. This concept aligns perfectly with our need to represent a ray of light, which has a point of origin and a direction of propagation.

Formally, we can define a ray $\mathbf{R}(t)$ in 3D space using a parametric equation:

$$\mathbf{R}(t) = \mathbf{O} + t\mathbf{D}$$

where:

- $\mathbf{O}$ is the origin point of the ray (a 3D vector)
- $\mathbf{D}$ is the direction vector of the ray (a 3D unit vector)
- t is a scalar parameter ($t \geq 0$)

This equation describes all points along the ray, starting from the origin and extending infinitely in the direction of $\mathbf{D}$. In practice, we often constrain t to an interval $[t\text{min}, t\text{max}]$ to define a specific segment of the ray.

2.2 Intersection Testing

Having established the mathematical and programmatic representation of a ray of light, we now turn our attention to determining how these rays interact with objects in our scene. Intersection testing is a crucial component of ray tracing, allowing us to simulate the behavior of light as it encounters various surfaces. To perform these tests efficiently, we leverage the geometric properties of vectors.

2.2.1 Mathematical Tools

Two fundamental vector operations are essential for intersection testing: the dot product and the cross product.

Dot Product The dot product of two vectors **A** and **B** is a scalar quantity defined as:

$$\mathbf{A} \cdot \mathbf{B} = AxBx + AyBy + AzBz$$

This operation has several useful properties for intersection testing:

- It can be used to calculate the angle between two vectors:

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|}$$

- It allows us to determine orthogonality (when the dot product is zero)
- It enables the computation of vector projections

In the context of ray tracing, the dot product is particularly useful for determining the angle between a ray and a surface normal, which is crucial for calculating reflection and refraction.

Cross Product The cross product of two vectors **A** and **B** results in a third vector **C** that is perpendicular to both:

$$\mathbf{C} = \mathbf{A} \times \mathbf{B} = (AyBz - AzBy, AzBx - AxBz, AxBy - AyBx)$$

The magnitude of the resulting vector is given by:

$$\|\mathbf{C}\| = \|\mathbf{A}\|\|\mathbf{B}\| \sin(\theta)$$

where θ is the angle between $\mathbf{A}$ and $\mathbf{B}$.

In intersection testing, the cross product serves several important purposes:

- It can be used to compute surface normals for triangles or polygons
- It helps in determining the orientation of surfaces relative to the ray
- It's useful in calculating barycentric coordinates for triangle intersection tests
- It can be employed to find perpendicular vectors, which is helpful in constructing coordinate systems for shading calculations

These mathematical tools form the foundation for implementing various intersection tests. For instance, when testing ray-triangle intersections, we can use the cross product to compute the triangle's normal and the dot product to determine if the ray is facing the correct side of the triangle. Similarly, for ray-plane intersections, the dot product helps us calculate the distance along the ray at which the intersection occurs.

In the following sections, we will explore how these tools are applied to specific geometric primitives, starting with spheres and planes, and then extending to more complex shapes and acceleration structures. By leveraging these vector operations, we can efficiently determine not only if a ray intersects an object, but also the exact point of intersection and the surface properties at that point, which are crucial for accurate light transport simulation in our path tracer.

2.2.2 Spheres

A sphere is a common geometric object in ray tracing, defined by its center $\mathbf{C}$ and radius R. To determine the intersection of a ray with a sphere, we substitute the ray equation $\mathbf{P}(t) = \mathbf{O}+t\mathbf{D}$ into the sphere's implicit equation:

$$\|\mathbf{P}(t) - \mathbf{C}\|^2 = R^2$$

Expanding and simplifying this equation yields:

$$\|\mathbf{O} + t\mathbf{D} - \mathbf{C}\|^2 = R^2$$

$$(\mathbf{O} - \mathbf{C} + t\mathbf{D}) \cdot (\mathbf{O} - \mathbf{C} + t\mathbf{D}) = R^2$$

$$(\mathbf{O} - \mathbf{C}) \cdot (\mathbf{O} - \mathbf{C}) + 2t\,(\mathbf{O} - \mathbf{C}) \cdot \mathbf{D} + t^2 \mathbf{D} \cdot \mathbf{D} = R^2$$

This is a quadratic equation in t:

$$t^2 \mathbf{D} \cdot \mathbf{D} + 2t\,(\mathbf{O} - \mathbf{C}) \cdot \mathbf{D} + \big((\mathbf{O} - \mathbf{C}) \cdot (\mathbf{O} - \mathbf{C}) - R^2\big) = 0$$

Letting $\mathbf{L} = \mathbf{O} - \mathbf{C}$, $a = \mathbf{D} \cdot \mathbf{D}$, $b = 2\mathbf{L} \cdot \mathbf{D}$, and $c = \mathbf{L} \cdot \mathbf{L} - R^2$, we solve the quadratic equation:

$$at^2 + bt + c = 0$$

The solutions for t are given by:

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

The discriminant $b^2 - 4ac$ determines the nature of the intersection:

- If $b^2 - 4ac < 0$, the ray does not intersect the sphere.
- If $b^2 - 4ac = 0$, the ray tangentially intersects the sphere at one point.
- If $b^2 - 4ac > 0$, the ray intersects the sphere at two points.

The following image illustrates a ray-sphere intersection:

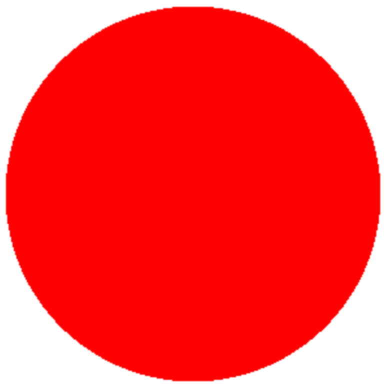

Figure 1: Ray-sphere intersection demonstration

In Figure 1, we can see a red circle representing a sphere on a white background. Notice the jagged pixelated border. This is because at one ray per pixel, a pixel can only be fully red (intersecting the sphere) or fully white (missing the sphere). We will fix this issue later.

2.2.3 Planes

A plane is defined by a point $\mathbf{P}_0$ on the plane and a normal vector $\mathbf{N}$. To find the intersection of a ray with a plane, we use the plane equation:

$$\mathbf{N} \cdot (\mathbf{P}(t) - \mathbf{P}_0) = 0$$

Substituting the ray equation $\mathbf{P}(t) = \mathbf{O} + t\mathbf{D}$:

$$\mathbf{N} \cdot (\mathbf{O} + t\mathbf{D} - \mathbf{P}_0) = 0$$

$$\mathbf{N} \cdot \mathbf{O} + t\,(\mathbf{N} \cdot \mathbf{D}) - \mathbf{N} \cdot \mathbf{P}_0 = 0$$

Solving for t:

$$t = \frac{\mathbf{N} \cdot (\mathbf{P}_0 - \mathbf{O})}{\mathbf{N} \cdot \mathbf{D}}$$

provided $\mathbf{N} \cdot \mathbf{D} \neq 0$. If $\mathbf{N} \cdot \mathbf{D} = 0$, the ray is parallel to the plane and does not intersect it.

The following image illustrates a ray-plane intersection:

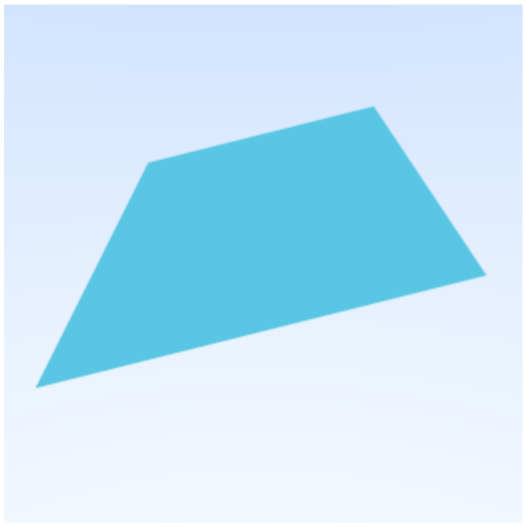

Figure 2: Ray-plane intersection demonstration

In Figure 2, we can see a ray intersecting a plane (represented as a quad for visualization purposes). The image demonstrates how a ray intersects the plane at a single point. This visual representation helps to understand the mathematical concept described above, where we solve for a single value of t to find the intersection point.

2.2.4 Extension to Object Meshes

Intersection tests can be extended to complex object meshes using triangle intersection algorithms. This is a crucial step in ray tracing, as most 3D models are represented as meshes composed of triangles.

Triangle Intersection UNNECESSARY TO DEMONSTRATE, REMOVE OR IMPROVE

The most common method for triangle intersection is the Möller–Trumbore algorithm. This algorithm is efficient and doesn't require precomputation of the triangle plane equation.

Given a ray $\mathbf{R}(t) = \mathbf{O} + t\mathbf{D}$ and a triangle defined by vertices $\mathbf{V}0$, $\mathbf{V}1$, and $\mathbf{V}_2$, we can express any point on the triangle as:

$$\mathbf{T}(u, v) = (1 - u - v)\mathbf{V}0 + u\mathbf{V}1 + v\mathbf{V}_2$$

where u and v are barycentric coordinates satisfying $u \geq 0$, $v \geq 0$, and $u + v \leq 1$.

The intersection point must satisfy:

$$\mathbf{O} + t\mathbf{D} = (1 - u - v)\mathbf{V}0 + u\mathbf{V}1 + v\mathbf{V}_2$$

This can be rewritten as a system of linear equations:

$$\begin{bmatrix} -\mathbf{D} & \mathbf{V}1 - \mathbf{V}0 & \mathbf{V}2 - \mathbf{V}0 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = \mathbf{O} - \mathbf{V}_0$$

Solving this system using Cramer's rule gives us t, u, and v. If $0 \leq u \leq 1$, $0 \leq v \leq 1$, $u + v \leq 1$, and $t > 0$, then the ray intersects the triangle.

Mesh Intersection For a mesh composed of many triangles, we perform the following steps:

For each triangle in the mesh: a. Perform the triangle intersection test. b. If an intersection is found, store the intersection point and distance.

After testing all triangles, return the closest intersection point (smallest positive t).

2.3 Placing Objects into the World

In computer graphics, assembling a scene involves not only defining the objects within it but also positioning them appropriately. Consider the classic demonstration scene known as the *Cornell Box* [3], which is a standard test for rendering algorithms to showcase accurate light transport and shading. From the previous sections, we've discussed how to test intersections with various geometric primitives. However, to build a complex scene like the Cornell Box, we need a systematic way to place and manipulate these objects in three-dimensional space.

Matrices are fundamental tools in linear algebra that we can use to efficiently represent and compute transformations in space. We can represent operations such as:

- **Translation** moves an object by a certain distance along the axes.
- **Rotation** turns an object around an axis passing through the origin.
- **Scaling** resizes an object by stretching or compressing it along the axes.

and put them all into mathematical objects that can be easily combined and applied to points and vectors in 3D space (and computed quickly on a GPU).

In our path tracer we can view a point or vector as a column vector (a matrix of N $\times$ 1):

$$\mathbf{v} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

Transforming this vector involves multiplying it by a transformation matrix:

$$\mathbf{v}' = \mathbf{M}\mathbf{v}$$

where $\mathbf{v}'$ is the transformed vector and $\mathbf{M}$ is the transformation matrix. If this transformation matrix is then applied to all points in our object, then we have successfully transformed our object.

2.3.1 Translation

We will start with translation as though it may appear the most simple: $\mathbf{v}' = \mathbf{v} + \mathbf{d}$ where $\mathbf{d} = \begin{bmatrix} dx \\ dy \\ dz \end{bmatrix}$ is the translation vector. Unfortunately, this breaks the laws of linearity (see subsection A.1).

This is a problem because translation cannot be represented with a 3×3 matrix in 3D space.

To unify translation with rotation and scaling in a single matrix operation, we introduce homogeneous coordinates. By adding an extra dimension to our vectors and matrices, we can represent all affine transformations, including translation, with 4×4 matrices.

In homogeneous coordinates, a point becomes:

$$\mathbf{v}_h = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

A translation matrix in homogeneous coordinates is:

$$\mathbf{T} = \begin{pmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The transformation is then applied via:

$$\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + T_x \\ y + T_y \\ z + T_z \\ 1 \end{pmatrix}$$

While we primarily use 3×3 matrices for rotation and scaling in our renderer, working with homogeneous coordinates and 4×4 matrices becomes essential for incorporating translation and perspective transformations, especially in more advanced rendering techniques.

2.3.2 Scaling

Scaling *is* a linear transformation which means that it can be represented using a 3×3 matrix, however as we would like all our transformations to multiply with each other we add a 4th scaling value of 1.

$$\begin{bmatrix} S_1 & 0 & 0 & 0 \\ 0 & S_2 & 0 & 0 \\ 0 & 0 & S_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} S_x \cdot x \\ S_y \cdot y \\ S_z \cdot z \\ 1 \end{pmatrix}$$

where S_x, S_y, and S_z are the scaling factors along the x, y, and z axes, respectively.

2.3.3 Rotation

Rotation can also be represented using a 3×3 matrix, but again, in practice they are represented as 4×4 matrices. Rotations in 3D are specified with an angle (in radians) and a rotation axis, with which you rotate the object around.

Rotation Around the X-axis Rotating a point around the x-axis by an angle θ:

$$\mathbf{R}_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation Around the Y-axis Rotating a point around the y-axis by an angle θ:

$$\mathbf{R}_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation Around the Z-axis Rotating a point around the z-axis by an angle θ:

$$\mathbf{R}_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

POTENTIALLY A SECTION HERE TALKING ABOUT PERFORMANCE AND TRANSFORMING VECTORS INTO MODEL SPACE

With these transformations we can now easily set up the scene like so:

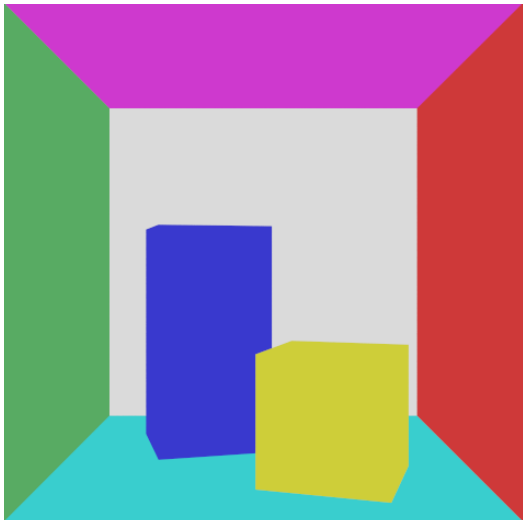

Figure 3: Cornell Box Objects

2.4 Color and Shading Models

Having established our scene and implemented the ability to compute intersections between rays and objects, we must find some way to use this information to simulate how light works in the real world. This involves modelling how light behaves upon hitting different surfaces.

Light and Material Interaction The appearance of objects in a scene is influenced by how they interact with light. When light strikes a surface, it can be absorbed, reflected, or transmitted. These interactions are governed by the material properties of the surface.

In the initial implementation 3, I simplified the shading process by assigning a static color to each object upon ray intersection. In reality a ray of light will take on the colour of a surfaces albedo, due to a partial reflection of all the wavelengths of white light.

This ray may then propagate through the scene, interacting with other objects and accumulating color contributions along its path. The material properties dictate whether the light is absorbed, reflected, or refracted at each interaction point. For example darker surfaces absorb more light and thus appear darker.

2.4.1 Diffuse Reflection

The first material I will cover is a perfectly diffuse one as it is the easiest to understand. If a ray hits the surface it will bounce randomly in any direction (with a little less brightness). This ray will then hit something else, and so on, and so on until it reaches a max recursion depth (at which point we can return black). This type of perfectly diffuse material is not the most realistic but is a good starting point, used by many of the first raytracing papers [9]. Here is what this method would look like with global illumination:

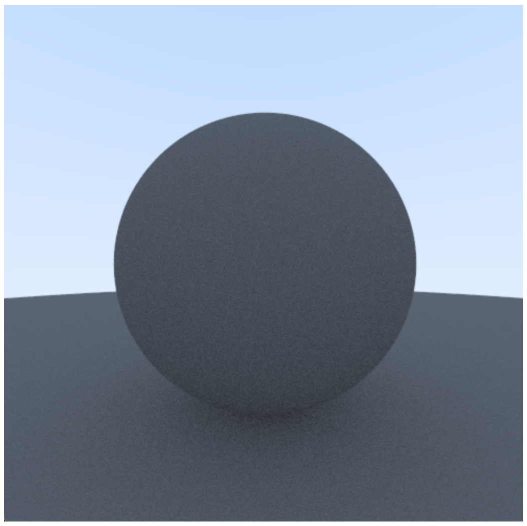

Figure 4: Perfectly diffuse material

Though this simple method yields good results, in reality, the Lambert Cosine Law comes into effect. This law states that the intensity of reflected light is dependent on the cosine of the angle between the incoming light direction and the surface normal.

Therefore, we need to adjust the power of reflection according to this principle. The intensity I of the reflected light is proportional to the cosine of the angle θ between the light direction $\mathbf{L}$ and the surface normal $\mathbf{N}$:

$$I = I_0 \cdot \max(\mathbf{L} \cdot \mathbf{N}, 0)$$

where I_0 is the intensity of the incoming light. This cosine dependency ensures that light hitting the surface at a shallow angle contributes less to the reflected intensity. This type of material is called a Lambertian:

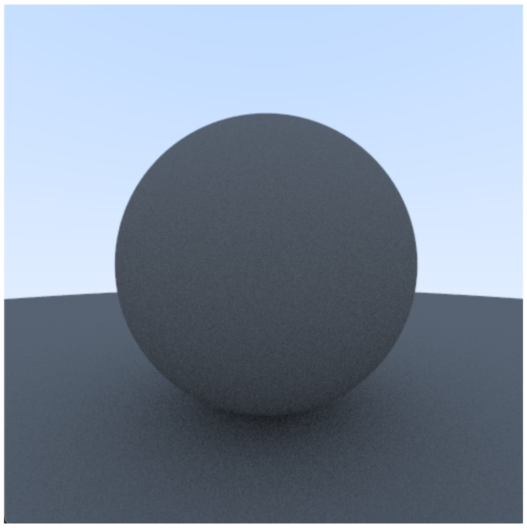

Figure 5: Lambertian diffuse material

If you look closely you will notice that the shadows are much more pronounced.

2.4.2 Specular Reflection

Specular reflection occurs on shiny surfaces where light is reflected in a specific direction. In a perfect mirror we want to "reflect" the ray according to its angle of incidence, instead of randomly in any direction. The reflection vector $\mathbf{R}$ is computed as:

$$\mathbf{R} = \mathbf{L} - 2(\mathbf{L} \cdot \mathbf{N})\mathbf{N}$$

The intensity of the reflected light depends on the angle between the reflection vector **R** and the view direction **V**:

$$I = I_0 \cdot \max(\mathbf{V} \cdot \mathbf{R}, 0)^n$$

where n is the shininess coefficient, determining the sharpness of the reflection.

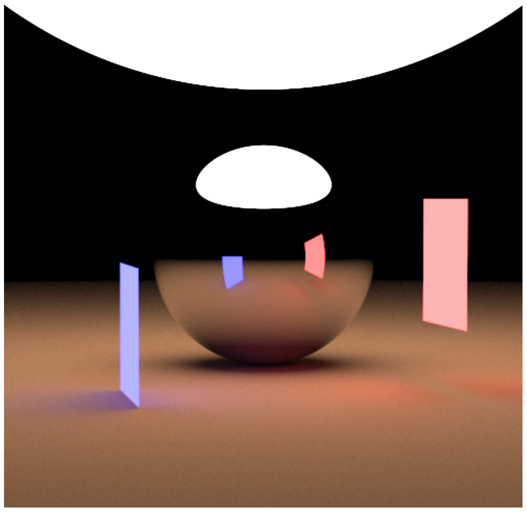

Figure 6: Specular material

Notice the bending of the lights in the spheres reflection.

3 Camera Dynamics

Cameras in path tracing simulate the human eye or a physical camera, capturing scenes from a specific viewpoint. The camera's parameters, such as position, orientation, and field of view, allowing us to play with the composition and visual realism of the final image.

Basis Vectors and Camera Orientation To accurately orient the camera in a scene, we create a local coordinate system using three orthonormal basis vectors: **u**, **v**, and **w**. These vectors allow us to easily project a viewport grid onto the physical pixels of our image. The vectors also allow us to move the camera's orientation and not have to worry about projection complexities.

The camera's orientation is specified by the following vectors:

w: This vector points from the camera position towards the look-at point, essentially defining the view direction. It is calculated as:

$$\mathbf{w} = \frac{\mathbf{C} - \mathbf{L}}{\|\mathbf{C} - \mathbf{L}\|}$$

where **C** represents the camera's position, and **L** is the focal point or the point the camera is directed towards.

u: This vector points to the camera's right and is derived from the cross product of the global up vector $\mathbf{v}_{\text{up}}$ and **w**:

$$\mathbf{u} = \frac{\mathbf{v}_{\text{up}} \times \mathbf{w}}{\|\mathbf{v}_{\text{up}} \times \mathbf{w}\|}$$

v: This vector points upward in the camera's local space, calculated by the cross product of **w** and **u**:

$$\mathbf{v} = \mathbf{w} \times \mathbf{u}$$

These vectors form a right-handed coordinate system, defining the camera's spatial alignment and enabling intuitive manipulation of the camera's orientation in the scene.

3.1 Perspective Projection

We currently have the ability to send rays throughout a 3D scene, but a screen is a 2D plane so we must project our data into it. This is achieved by calculating a viewports dimensions from our camera's basis vectors, alongside pixel information from our applications window (I am using GLFW to manage the window).

To simulate the projection of rays from the camera through each pixel on the image plane, we calculate the position of each pixel within the camera's local coordinate system. The pixel positions represent rays emanating from the camera through the viewport, defined in terms of the basis vectors **u** and **v**.

the viewport dimensions based on the vertical field of view (FOV) and the aspect ratio of the rendered image. The viewport height is computed using the trigonometric relationship:

$$\text{viewportHeight} = 2 \cdot \tan\left(\frac{\text{vertFOV}}{2}\right) \cdot \text{focalLength}$$

The viewport width is then set according to the aspect ratio:

$$\text{viewportWidth} = \text{viewportHeight} \cdot \frac{\text{imageWidth}}{\text{imageHeight}}$$

By doing this, we maintain the proportions of the scene and prevent distortion in the final rendered image.

With the basis vectors established, we define the horizontal viewportU and vertical viewportV extents of the viewport:

$$\text{viewportU} = \text{viewportWidth} \cdot \mathbf{u}$$

$$\text{viewportV} = \text{viewportHeight} \cdot \mathbf{v}$$

These vectors represent the physical dimensions of the viewport in the camera's local coordinate system. To calculate the movement from one pixel to the next across the image, we determine the pixel spacing (delta vectors) in both horizontal and vertical directions:

$$\text{pixelDeltaU} = \frac{\text{viewportU}}{\text{imageWidth}}$$

$$\text{pixelDeltaV} = \frac{\text{viewportV}}{\text{imageHeight}}$$

Finally, we determine the position of the top-left corner of the viewport viewportUpperLeft by offsetting the camera center by the focal length along **w** and half the viewport dimensions along **u** and **v**:

$$\text{viewportUpperLeft} = \mathbf{C} - (\text{focalLength} \cdot \mathbf{w}) - \frac{\text{viewportU}}{2} - \frac{\text{viewportV}}{2}$$

The coordinates for the first pixel (the upper-left corner, pixelTL) are then calculated by adjusting this position by half a pixel's width and height:

$$\text{pixelTL} = \text{viewportUpperLeft} + 0.5 \cdot (\text{pixelDeltaU} + \text{pixelDeltaV})$$

Iterating over the rest of the pixels in the scene is a simple matter of adding multiples of pixelDeltaU + pixelDeltaV to the top left pixel.

Field Of View The camera's field of view (FOV) determines how much of the scene is visible through the viewport, affecting the perceived scale and perspective:

The vertical field of view (vertFov) defines the camera's vertical range:

$$\text{viewportHeight} = 2 \cdot \tan\left(\frac{\text{vertFov}}{2}\right)$$

- The viewport width is adjusted based on the aspect ratio (aspectRatio):

$$\text{viewportWidth} = \text{aspectRatio} \cdot \text{viewportHeight}$$

These parameters ensure that the scene is rendered with the correct proportions and perspective, replicating how a real-world camera captures a scene. We can also manipulate the FOV to get some artistic effects:

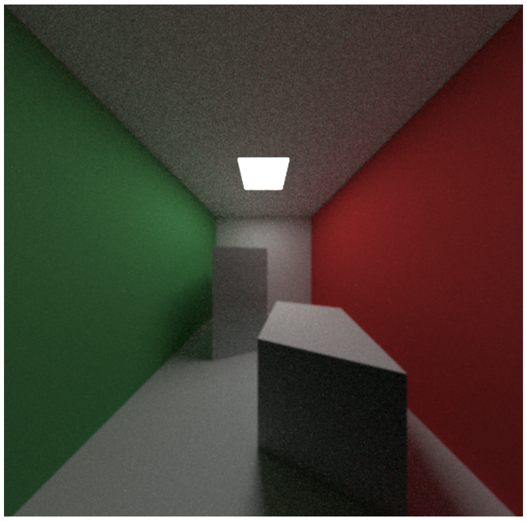

Figure 7: Cornell box rendered with high FOV

The geometry of the Cornell box has remained identical, however we have achieved an effective emulation of a "Fish-eye" lens.

3.2 Camera Aliasing

The camera model is fundamental to the ray tracing process, simulating the viewpoint from which the scene is observed. It generates rays from the camera's origin, passing through each pixel on the image plane:

A crucial challenge in rendering involves aliased edges, where a pixel either hits an object or misses it entirely. This binary decision creates jagged edges, as briefly referenced in the aliased interaction in Figure 8.

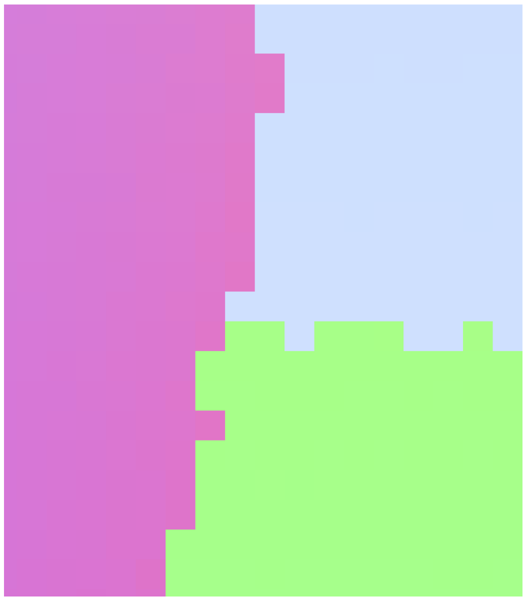

Figure 8: Aliased interactions

To address this, we employ anti-aliasing techniques. One effective method is Super Sampling Anti-Aliasing (SSAA), where multiple samples per pixel are taken, and their colors are averaged. By randomly choosing a direction within the pixel, we smooth out the transitions and enhance image quality:

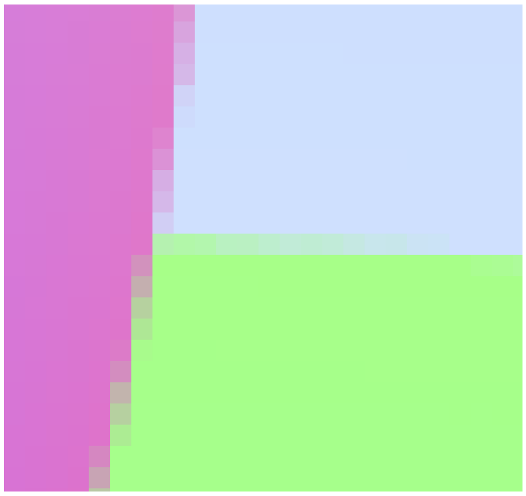
Figure 9: SSAA intersections

SSAA significantly reduces aliasing artifacts, producing more realistic and visually appealing results by blurring the sharp transitions at object boundaries. The significant downside is that SSAA is incredibly expensive to perform, if we want 16xSSAA then our computation time will also 16x.

4 Acceleration Structures

4.1 Bounding Volume Hierarchies (BVH)

4.1.1 Overview of BVH

An advanced structure for efficient intersection tests.

4.1.2 Technical Implementation

- **Efficient Ray-Object Intersection Tests:** Using BVH for faster intersection calculations.
- **Building and Traversing BVH:** Methods for constructing and navigating BVH structures.

4.1.3 Algorithmic Analysis

Performance analysis of BVH algorithms for various scene complexities.

5 Monte Carlo Methods for Ray Tracing

5.1 Probability and Monte Carlo Methods

5.1.1 Improved Random Sampling

Stochastic methods improve sampling efficiency in rendering.

5.1.2 Monte Carlo Integration

Using random sampling to approximate integrals in light transport.

5.1.3 Importance Sampling

Advanced sampling techniques enhance efficiency and image quality. reference 3.3

5.2 Integration for Lighting Models

5.2.1 Radiance and Light Transport Equations

Mathematical models for light behavior in rendering.

5.2.2 Numerical Integration Techniques

Applying numerical methods to solve lighting equations.

5.2.3 Technical Implementation

Implementing global illumination and integrating direct and indirect lighting.

6 Technical Description of the System

Put prerequistes here, opengl cpp etc. NOT TRUE come up with a MORE ACCURATE DESCRIPTION In this section, we will focus on how the discussed ele

6.1 System Architecture

6.1.1 Design Principles Behind the Modular Architecture

I have found that when architechting a complex system it is essential to keep several non-functional attributes in mind and to always assess your architectural decisions against these. A path tracer requires several key non-functional attributes to be effective and maintainable (ordered by descending importance):

1. Performance: The system must be able to handle complex scenes and produce high-quality renders in a reasonable time frame.
2. Extensibility: As rendering techniques evolve, the system should be able to incorporate new algorithms and features easily.
3. Maintainability: The codebase should be easy to understand, debug, and modify.

With these requirements in mind it naturally leads to a modular monolith approach. I find that Modular design is suited to our non-functional attributes as:

- Encapsulation of Functionality: Each module encapsulates a specific set of related functions, allowing for optimized performance within modules and clear interfaces between them.
- Loose Coupling: Modules interact through well-defined interfaces, making it easier to modify or replace individual components without affecting the entire system (imagine you want to switch to the Vulkan API, this should be quite an easy task!).
- Parallel Development: Different teams or developers can work on separate modules simultaneously, speeding up development and integration of new features.
- Testability: Individual modules can be tested in isolation, facilitating more thorough and efficient testing processes.

The modular monolith approach strikes a balance between the simplicity of a monolithic architecture and the flexibility of a fully distributed system. This approach is particularly well-suited for a path tracer, where tight integration between components is necessary for performance, but clear separation of concerns is crucial for ongoing development and maintenance.

6.1.2 Overview of Core Components and Their Interactions

The following C4 diagram provides an overview of the path tracer's modules, along with their responsibilities and interactions:

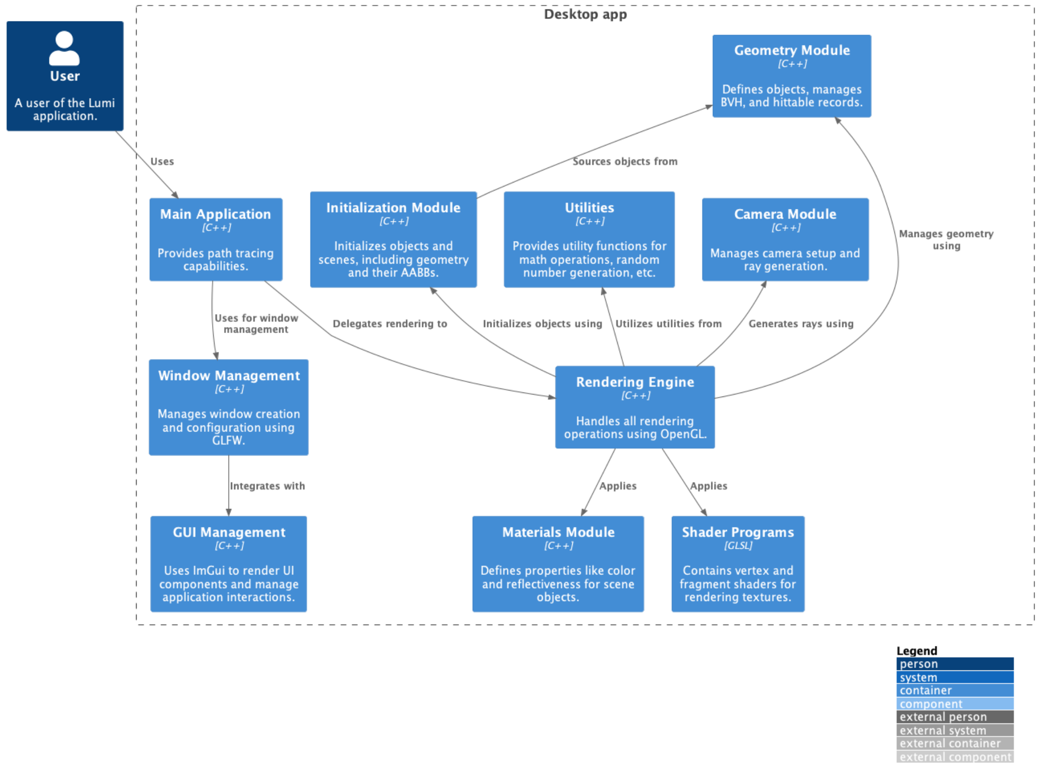
Desktop app
User
A user of the Lumi application.
Uses
Main Application
[C++]
Provides path tracing capabilities.
Initialization Module
[C++]
Initializes objects and scenes, including geometry and their AABBs.
Utilities
[C++]
Provides utility functions for math operations, random number generation, etc.
Camera Module
[C++]
Manages camera setup and ray generation.
Geometry Module
[C++]
Defines objects, manages BVH, and hittable records.
Sources objects from
Manages geometry using
Uses for window management
Delegates rendering to
Initializes objects using
Utilizes utilities from
Generates rays using
Window Management
[C++]
Manages window creation and configuration using GLFW.
Rendering Engine
[C++]
Handles all rendering operations using OpenGL.
Integrates with
Applies
Applies
GUI Management
[C++]
Uses ImGui to render UI components and manage application interactions.
Materials Module
[C++]
Defines properties like color and reflectiveness for scene objects.
Shader Programs
[GLSL]
Contains vertex and fragment shaders for rendering textures.
Legend
person
system
container
component
external person
external system
external container
external component

Figure 10: C4 diagram of the path tracer system architecture

The Rendering Engine acts as the central hub, coordinating the activities of other modules. It delegates specific tasks to specialized modules (e.g., ray generation to the Camera Module, object intersection tests to the Geometry Module) and applies materials and shaders as needed.

6.1.3 Main Control Loop's Role in Managing Rendering Operations

Here talk about call stack, main loop is the top level which sets up stuff, and summons the renderer to start. The main control loop orchestrates rendering, coordinating ray generation, shading, and image synthesis.

6.2 Core Components

Geometry: An Abstraction for Hittable Objects Provides a unified interface for diverse geometrical shapes.

Bounding Structures: Use of BVH for Efficient Intersection Tests Employs BVH for accelerated intersection tests, enhancing performance.

Scene Manager: Object Storage and Scene Graph Management The Scene Manager organizes objects in a scene graph, optimizing rendering operations.

Renderer: Ray Generation and Intersection Handling Handles ray generation and calculates light interactions with surfaces.

Camera Module: Virtual Camera Settings and Primary Ray Generation Simulates camera properties and generates primary rays for rendering.

6.3 Utilising OpenGL for Visualising Convergence

In traditional path tracing implementations, the final image is typically output once the required number of samples per pixel has been reached. However, this approach doesn't provide insight into the convergence process. To address this limitation, we've implemented a progressive rendering system using OpenGL, allowing for real-time visualization of the path tracing process.

Progressive Rendering The core of this visualization technique is the accumulation buffer. Instead of directly outputting the final pixel colors, we continuously update an accumulation buffer:

```
vector<vec3> accumulationBuffer(IMAGE_SIZE, 0);
vector<int> sampleCount(IMAGE_SIZE, 0);
```

The `accumulationBuffer` stores the sum of all samples for each pixel, while `sampleCount` keeps track of the number of samples per pixel.

The rendering process is modified to update the accumulation buffer after each sample:

```
accumulationBuffer[index] += rayColor(r, world, maxDepth);
sampleCount[index] += 1;
```

This incremental approach enables us to visualize the gradual convergence of the image over time. To display the progressively rendered image, we utilize OpenGL's texture mapping capabilities. The current state of the render is mapped to a texture that covers the entire window:

```
void updateTexture(unsigned int texture, const vector<vec3> &image,
const int &WIDTH, const int &HEIGHT)
    {
        glBindTexture(GL_TEXTURE_2D, texture);
        glTexSubImage2D(WIDTH, HEIGHT, image.data());
    }
```

MENTION MAPPING TO A UNIFORM 2D IN FRAGMENT SHADER AND OPENGL PIPELINE HERE This texture is then rendered onto a quad that fills the entire screen, allowing for real-time updates of the rendering progress.

Benefits of Real-time Visualization This approach offers several advantages:

- Parameter Tuning: Real-time visualization allows for quick iterations when adjusting rendering parameters, as the effects of changes are immediately visible.

- Debugging Aid: It becomes easier to identify and debug issues in the path tracing algorithm as they manifest visually during rendering.
- Progress Monitoring: Users can make informed decisions about when to terminate the rendering based on the current image quality.

Architecture Considerations The integration of OpenGL for visualization required careful consideration of the system architecture:

- Separation of Concerns: The core path tracing logic remains separate from the visualization code, maintaining modularity.
- Efficient Data Transfer: The accumulation buffer serves as an intermediate representation, minimizing data transfer between the CPU and GPU. MENTION HOW SLOW IT IS/ CPU PATH TRACING VS
- Flexible Rendering Pipeline: The system is designed to allow easy switching between progressive rendering and traditional batch rendering if needed. CODE HERE?

By leveraging OpenGL for real-time visualization, we've enhanced the path tracer's utility as both a rendering tool and a development aid, providing deeper insights into the rendering process and facilitating more efficient algorithm refinement.

7 Conclusion

7.1 Summary of Key Points

Monte Carlo methods effectively enhance path tracing and image synthesis, improving quality and efficiency.

7.2 Future Work

Future exploration of advanced sampling strategies and optimizations for real-time rendering.

7.3 Final Thoughts

Statistical methods are crucial for high-quality rendering solutions.

8 References

References

[1] Per Christensen, George Harker, Jonathan Shade, Brenden Schubert, and Dana Batali. Multiresolution radiosity caching for efficient preview and final quality global illumination in movies. *ACM Transactions on Graphics (TOG)*, 37(5):1–12, 2018.

[2] Andrew S. Glassner, editor. *An Introduction to Ray Tracing.* Elsevier, 1989.

[3] Henrik Wann Jensen. Cornell box. `https://graphics.stanford.edu/~henrik/images/cbox.html`. Accessed: 21 Oct 2024.

[4] James T. Kajiya. The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, pages 143–150, 1986.

[5] Alexander Keller, Luca Fascione, Marcos Fajardo, Iliyan Georgiev, Per Christensen, Johannes Hanika, Christian Eisenacher, Greg Nichols, and Toshiya Hachisuka. The path tracing revolution in the movie industry. In *ACM SIGGRAPH 2015 Courses*, pages 1–7. ACM, 2015.

[6] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation.* Morgan Kaufmann, 2016.

[7] Christoph Schied, Anton Kaplanyan, Chris Wyman, Anjul Patney, Chakravarty R Alla Chaitanya, John Burgess, Shiqiu Liu, Carsten Dachsbacher, Aaron Lefohn, and Marco Salvi. Spatiotemporal variance-guided filtering: real-time reconstruction for path-traced global illumination. In *Proceedings of High Performance Graphics*, pages 1–12, 2017.

[8] Eric Veach. *Robust Monte Carlo Methods for Light Transport Simulation.* PhD thesis, Stanford University, 1997.

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

A Detailed Mathematical Derivations

A.1 Linearity of Transformations

To demonstrate why translation is not a linear transformation, we consider the following properties:

Law of Additivity: $T(\mathbf{u}+\mathbf{v}) = T(\mathbf{u}) + T(\mathbf{v})$
Law of Homogeneity: $T(c\mathbf{u}) = cT(\mathbf{u})$
If we test the linearity conditions:
Additivity:

$$T(\mathbf{u}+\mathbf{v}) = (\mathbf{u}+\mathbf{v}) + \mathbf{b} \neq T(\mathbf{u}) + T(\mathbf{v}) = (\mathbf{u}+\mathbf{b}) + (\mathbf{v}+\mathbf{b})$$

The right-hand side becomes $\mathbf{u}+\mathbf{v}+2\mathbf{b}$, which is not equal to the left-hand side.
Homogeneity:

$$T(c\mathbf{u}) = c\mathbf{u} + \mathbf{b} \neq cT(\mathbf{u}) = c(\mathbf{u}+\mathbf{b}) = c\mathbf{u} + c\mathbf{b}$$

The translation vector $\mathbf{b}$ remains unchanged and doesn't scale with c, breaking this property as well.
Thus, the translation is an affine transformation, not a linear transformation, because it doesn't preserve these linearity conditions.

B Code Snippets and Pseudocode

C Additional Figures and Diagrams