Path Tracing Renderer Using Monte Carlo Methods

Silas Maughan

October 7, 2024

Abstract

This report presents a study on the 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. The report discusses the mathematical foundations, implementation details, and performance evaluations of different Monte Carlo sampling strategies and variance reduction techniques.

Contents

1 Introduction

1.1 Purpose and Scope

1.1.1 Goals of the Document

The goal of this document is to explain the implementation of a path tracing renderer using Monte Carlo methods, focusing on different sampling techniques and variance reduction methods to enhance image quality and rendering efficiency.

1.1.2 Overview of the Methodological Approach

The methodological approach will be outlined and the effectiveness of these techniques will be evaluated through experimental results.

1.2 Background

1.2.1 Overview of Ray Tracing

Path tracing is a rendering technique used to create realistic images by simulating the way light interacts with objects in a scene. Unlike traditional ray tracing, which traces a single path of light from the eye to the light source, path tracing traces multiple light paths to account for complex interactions like reflection, refraction, and scattering. This report details the implementation of a path tracing renderer using Monte Carlo integration to approximate the rendering equation.

1.2.2 Importance in Computer Graphics

Path tracing is crucial in computer graphics for achieving photo-realistic rendering, as it accurately simulates the physical behavior of light.

2 Ray Tracing Fundamentals

2.1 Rays

2.1.1 Definition of a Ray

A ray in 3D graphics is an essential construct for simulating the propagation of light through a scene. It is defined by an origin point $\mathbf{O}$ and a direction vector $\mathbf{D}$. Mathematically, we can express a point $\mathbf{P}(t)$ along the ray as:

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

where t is a real number parameter. This equation indicates that the ray starts at the origin $\mathbf{O}$ and extends infinitely in the direction of $\mathbf{D}$. In practical terms, t is typically constrained within an interval $[t_{\min}, t_{\max}]$ to define a segment of the ray.

The geometric interpretation of a ray is straightforward: it represents a half-line extending from $\mathbf{O}$. This formulation is crucial for tracing the path of light as it interacts with objects in a scene.

2.1.2 Ray-Object Intersection

At the core of ray tracing is the need to determine whether a ray intersects an object in the scene. This involves solving for t when a point $\mathbf{P}(t)$ on the ray lies on the surface of the object. Each type of object requires a specific intersection test, leading to a variety of mathematical formulations.

2.2 Vectors and Their Operations

Vectors play a pivotal role in ray tracing, representing points, directions, normals, and colors. A 3D vector $\mathbf{A}$ can be represented as (A_x, A_y, A_z). We will cover essential vector operations that are fundamental to ray tracing.

2.2.1 Dot Product

The dot product of two vectors $\mathbf{A}$ and $\mathbf{B}$ is a scalar quantity given by:

$$\mathbf{A} \cdot \mathbf{B} = A_x B_x + A_y B_y + A_z B_z$$

The dot product is used to calculate the angle between two vectors, determine orthogonality, and compute projections. For example, the angle θ between $\mathbf{A}$ and $\mathbf{B}$ can be found using:

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

2.2.2 Cross Product

The cross product of two vectors $\mathbf{A}$ and $\mathbf{B}$ results in a third vector $\mathbf{C}$ that is perpendicular to both:

$$\mathbf{C} = \mathbf{A} \times \mathbf{B} = (A_y B_z - A_z B_y, A_z B_x - A_x B_z, A_x B_y - A_y B_x)$$

The magnitude of $\mathbf{C}$ is given by:

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

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

2.3 Intersection Calculations

2.3.1 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.

2.3.2 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.

2.4 Color and Shading Models

2.4.1 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.

2.4.2 Diffuse Reflection

A perfectly diffuse (Lambertian) surface scatters incident light uniformly in all directions. 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.

2.4.3 Specular Reflection

Specular reflection occurs on shiny surfaces where light is reflected in a specific 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 $\mathbf{R}$ and the view direction $\mathbf{V}$:

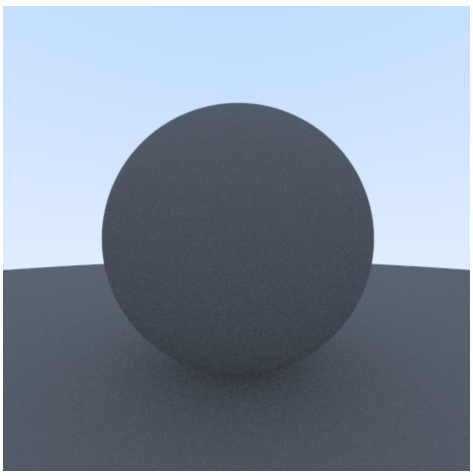
$$I = I_0 \cdot \max(\mathbf{V} \cdot \mathbf{R}, 0)^n$$

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

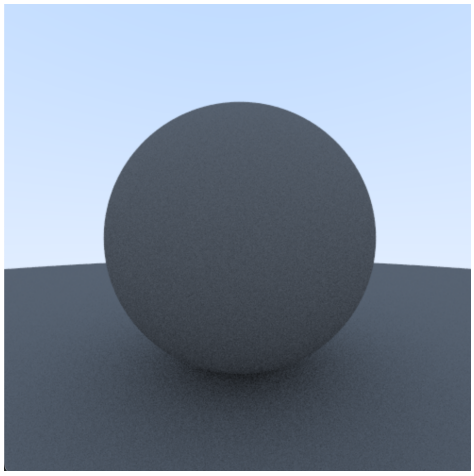
2.4.4 Technical Implementation

Color models in code often represent colors as vectors with three components: red, green, and blue (RGB). Each component ranges from 0 to 1.

Lambertian Reflectance: In a Lambertian model, the scattering of rays is uniformly distributed over the hemisphere centered on the surface normal. This can be implemented using Monte Carlo techniques to randomly sample directions within the hemisphere.



(a) Rendered image using Uniform Diffuse Renderer



(b) Rendered image using Lambertian Diffuse Renderer

Figure 1: Comparison of rendering techniques

Specular Reflectance: For specular materials, the direction of the reflected ray is computed using the reflection equation. This is crucial for simulating shiny surfaces such as metals and mirrors.

2.5 Linear Transformations

2.5.1 Translation, Rotation, and Scaling

Objects in a 3D scene can be manipulated using transformations such as translation, rotation, and scaling. These transformations are represented by matrices.

Translation: A translation matrix $\mathbf{T}$ moves an object by a vector $\mathbf{d} = (d_x, d_y, d_z)$:

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

Rotation: A rotation matrix $\mathbf{R}$ rotates an object around an axis. For

example, a rotation around the z-axis by an angle θ is given by:

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

Scaling: A scaling matrix $\mathbf{S}$ scales an object by factors s_x, s_y, and s_z:

$$\mathbf{S} = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

2.5.2 Matrix Representations

Transformation matrices allow for the linear transformation of vectors. Given a vector $\mathbf{v}$ and a transformation matrix $\mathbf{M}$, the transformed vector $\mathbf{v}'$ is:

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

Combining multiple transformations is achieved by matrix multiplication.

2.5.3 Technical Implementation

Transforming Objects and Rays: Applying matrix operations to objects and rays in the scene ensures consistent transformations. The composite matrix $\mathbf{M}$ for a series of transformations is:

$$\mathbf{M} = \mathbf{TRS}$$

Applying Transformations to the Scene: All elements in the scene, including geometry and light sources, are transformed using composite matrices to maintain spatial coherence.

3 Advanced Techniques in Ray Tracing

3.1 Camera and Viewing

3.1.1 Camera Model and Rays

The camera in ray tracing simulates the viewpoint from which the scene is rendered. It projects rays from its origin through each pixel on the image

plane, capturing the scene from a specific perspective. The mathematical foundation underlying the camera's functionality involves various concepts from linear algebra.

Basis Vectors and Camera Orientation

Basis Vectors The camera's orientation is defined by three orthonormal basis vectors: $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$.

- $\mathbf{w}$: Vector pointing from the camera position to the view direction. It is computed as:
$$\mathbf{w} = \frac{\mathbf{C} - \mathbf{L}}{\|\mathbf{C} - \mathbf{L}\|}$$
where $\mathbf{C}$ is the camera position and $\mathbf{L}$ is the focal point.

- $\mathbf{u}$: Vector pointing to the right of the camera, computed using the cross product of the up vector $\mathbf{v}_{\text{up}}$ and $\mathbf{w}$:
$$\mathbf{u} = \frac{\mathbf{v}_{\text{up}} \times \mathbf{w}}{\|\mathbf{v}_{\text{up}} \times \mathbf{w}\|}$$

- $\mathbf{v}$: Vector pointing upwards relative to the camera, computed as:
$$\mathbf{v} = \mathbf{w} \times \mathbf{u}$$

These basis vectors form a right-handed coordinate system and define the camera's orientation.

Viewport and Field of View The camera's field of view (FOV) determines the extent of the observable world. The vertical FOV (vertFov) is the angle of the observable world in the vertical direction:

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

The viewport width is adjusted according to the aspect ratio (aspectRatio):

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

Perspective Projection To generate rays from the camera through each pixel, we calculate the position of each pixel in the camera's coordinate system. The pixel positions are offset from a reference point (the upper-left corner of the viewport) by the basis vectors **u** and **v**.

Implementation in Camera Class The 'Camera' class encapsulates the logic for positioning and orienting the camera, as well as generating rays through the pixels.

```
class Camera
{
public:
    Camera(unsigned int imageWidth, unsigned int imageHeight, const glm::vec3 &pos
    Camera(unsigned int imageWidth, unsigned int imageHeight, const CamPos &camPos

    void render(const HittableList &world, std::vector<glm::vec3> &accumulationBuf

    int image_width;
    int image_height;

private:
    glm::vec3 center;
    glm::vec3 look_at;
    glm::vec3 vup;
    float vert_fov;
    int sample_per_pixel_per_frame;
    int sqrt_sample_per_pixel_per_frame;
    float recip_sqrt_sppf;
    int max_depth;

    glm::vec3 pixel00_loc;
    glm::vec3 pixel_delta_u;
    glm::vec3 pixel_delta_v;
    glm::vec3 u;
    glm::vec3 v;
    glm::vec3 w;

    void initialize();
```

```
    Ray getRandomRay(int x, int y) const;
    Ray getRandomStratifiedRay(glm::vec3 pixelCenter, int gridX, int gridY) const;
    glm::vec3 rayColor(const Ray &r, const HittableList &world, int depth) const;
};
```

The ‘Camera‘ class includes member variables to store the camera's position, orientation, and configuration parameters.

Initialization and Basis Vector Calculation The ‘initialize‘ method sets up the camera's coordinate system using the basis vectors:

```
void Camera::initialize() {
    float theta = glm::radians(vert_fov);
    float h = tan(theta / 2);
    float viewport_height = 2.0f * h;
    float viewport_width = viewport_height * float(image_width) / float(image_heig

    w = glm::normalize(center - look_at);
    u = glm::normalize(glm::cross(vup, w));
    v = glm::cross(w, u);

    pixel00_loc = center - (u * viewport_width / 2.0f) - (v * viewport_height / 2.
    pixel_delta_u = u * viewport_width / float(image_width);
    pixel_delta_v = v * viewport_height / float(image_height);
}
```

Here,

- **w** is calculated as the normalized vector from the camera position to the look-at point.
- **u** is the normalized cross product of the up vector and **w**.
- **v** is the cross product of **w** and **u**.

Ray Generation The method ‘getRandomRay‘ generates rays from the camera through each pixel, considering anti-aliasing by randomly offsetting the ray within the pixel:

```
Ray Camera::getRandomRay(int x, int y) const {
    float u_offset = (x + random_float()) * recip_sqrt_sppf;
    float v_offset = (y + random_float()) * recip_sqrt_sppf;
    glm::vec3 ray_origin = center;
    glm::vec3 ray_direction = glm::normalize(pixel00_loc + u_offset * pixel_delta_
    return Ray(ray_origin, ray_direction);
}
```

This method uses the basis vectors to compute the direction of the rays passing through each pixel, incorporating random offsets to achieve anti-aliasing.

Rendering The 'render' method generates rays for each pixel and computes the color by tracing these rays through the scene. The final image is accumulated and averaged over multiple samples for anti-aliasing.

```
void Camera::render(const HittableList &world, std::vector<glm::vec3> &accumulatio
    for (int y = 0; y < image_height; ++y) {
        for (int x = 0; x < image_width; ++x) {
            glm::vec3 color(0.0f);
            for (int s = 0; s < sample_per_pixel_per_frame; ++s) {
                Ray r = getRandomRay(x, y);
                color += rayColor(r, world, max_depth);
            }
            accumulationBuffer[y * image_width + x] += color;
            sampleCount[y * image_width + x] += sample_per_pixel_per_frame;
            glm::vec3 averagedColor = accumulationBuffer[y * image_width + x] / fl
            // Store averagedColor in the image buffer
        }
    }
}
```

This process ensures that each pixel's color is the result of multiple samples, reducing aliasing and producing a smoother image.

Geometric Interpretation and Linear Transformations

Changing Basis The camera's view can be thought of as a change of basis. The world coordinates are transformed into the camera's coordinate system defined by the basis vectors **u**, **v**, and **w**.

Linear Transformations Linear transformations, represented by matrices, are used to position and orient the camera. The transformation from world coordinates to camera coordinates involves translating the camera to the origin and rotating the basis vectors to align with the standard axes.

Concept of Vector Space The camera's coordinate system forms a vector space where the basis vectors **u**, **v**, and **w** span the space. The transformations applied to these vectors ensure linear independence, meaning they form a valid coordinate system for the camera.

Systems of Linear Equations In rendering, solving systems of linear equations can be used to determine intersections of rays with objects, transformations, and more. These systems can be represented and solved using matrix methods and vector operations.

In conclusion, the 'Camera' class in ray tracing leverages concepts from linear algebra such as basis vectors, linear transformations, and vector spaces to simulate the projection of rays through an image plane. This allows for accurate positioning, orientation, and rendering of the scene from the camera's perspective.

3.2 Acceleration Structures

3.2.1 Bounding Volume Hierarchies (BVH)

An advanced structure for efficient intersection tests.

3.2.2 Technical Implementation

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

3.3 Anti-Aliasing and Sampling

3.3.1 Aliasing Problems in Rendering

Understanding the impact of aliasing and methods to mitigate it.

3.3.2 Supersampling and Adaptive Sampling

Techniques to enhance image quality.

3.3.3 Technical Implementation

- **Implementing Supersampling:** Methods for applying supersampling in rendering.

4 Mathematical Methods for Ray Tracing

4.1 Probability and Monte Carlo Methods

4.1.1 Random Sampling Techniques

Utilizing stochastic methods to improve sampling efficiency.

4.1.2 Basic Probability Concepts

Foundational principles applicable to Monte Carlo methods.

4.1.3 Monte Carlo Integration

Using random sampling to approximate integrals.

4.1.4 Technical Implementation

- **Path Tracing:** Implementing Monte Carlo techniques in path tracing.
- **Importance Sampling and Russian Roulette:** Advanced Monte Carlo techniques to enhance efficiency.

4.2 Integration for Lighting Models

4.2.1 Radiance and Light Transport Equations

Mathematical models for light behavior.

4.2.2 Numerical Integration Techniques

Applying numerical methods to solve lighting equations.

4.2.3 Technical Implementation

- **Implementing Global Illumination:** Techniques for comprehensive lighting calculations.
- **Integrating Direct and Indirect Lighting:** Methods to account for all lighting contributions.

5 Technical Description of the System

5.1 System Architecture

5.1.1 Overview of Core Components and Their Interactions

The path tracer is a sophisticated system that includes several core components, each playing a critical role in the rendering pipeline. The seamless interaction between these components ensures efficient and accurate scene rendering. Key components include the Scene Manager, Camera Module, Renderer, and various support structures. Together, they provide a framework that allows for the simulation of realistic lighting and shadows, handling complex geometries with ease.

5.1.2 Main Control Loop's Role in Managing Rendering Operations

At the heart of the path tracer lies the main control loop, which orchestrates the entire rendering process. It is responsible for coordinating the activities of individual components, ensuring that tasks such as ray generation, intersection testing, shading, and image synthesis are performed in a synchronized manner. This loop not only manages resource allocation and timing but also

adapts to dynamic changes within the scene, thus enabling a robust rendering workflow.

5.1.3 Design Principles Behind the Modular Architecture

The system's architecture is founded on modular design principles, which facilitate ease of maintenance and scalability. By encapsulating functionalities within discrete modules, the architecture allows for independent development and testing. This modularity also supports the integration of new features and optimizations with minimal disruption to existing functionalities, ensuring that the system remains adaptable and forward-compatible.

5.2 Core Components

5.2.1 Scene Manager: Object Storage and Scene Graph Management

The Scene Manager serves as the repository for all objects within the scene, maintaining an organized structure through a scene graph. This graph efficiently tracks hierarchical relationships and transformations, enabling complex operations such as culling and level-of-detail adjustments. By optimizing data retrieval and updates, the Scene Manager ensures that rendering processes are both swift and accurate.

5.2.2 Camera Module: Virtual Camera Settings and Primary Ray Generation

The Camera Module is tasked with simulating the properties of a virtual camera, defining parameters such as field of view, aperture, and focus distance. It is responsible for generating primary rays that initiate the path tracing process, translating the camera's perspective into digital commands that guide subsequent rendering stages.

5.2.3 Renderer: Ray Generation and Intersection Handling

The Renderer is the engine of the path tracing system, where rays are generated and intersections with scene objects are meticulously handled. It implements algorithms that determine the paths of light as they interact with

surfaces, calculating interactions such as reflection, refraction, and absorption. This component ensures that every detail of light behavior is captured, contributing to the generation of photorealistic images.

5.2.4 Geometry: An Abstraction for Hittable Objects

The Geometry component abstracts the complexities of various hittable objects within the scene. By providing a unified interface for diverse geometrical shapes, it facilitates efficient intersection tests and material interactions. This abstraction ensures that developers can introduce new shapes or modify existing ones without altering the core rendering logic.

5.2.5 Bounding Structures: Use of BVH for Efficient Intersection Tests

Bounding Volume Hierarchies (BVH) are employed as acceleration structures to enhance the efficiency of intersection tests. By organizing objects into a tree structure, BVH reduces the number of checks required to find intersections, thereby accelerating rendering times. This strategy is crucial in handling complex scenes with numerous objects, ensuring that performance remains optimal.

5.2.6 Texture Mapping: Mapping the Progressively Converging Image to an OpenGL Texture by Using OpenGL Shaders

Texture Mapping involves applying textures to 3D models, enhancing their surface detail and realism. Using OpenGL shaders, textures are dynamically mapped to objects as the rendered image progressively converges. This technique leverages the power of GPU processing, allowing for real-time updates and intricate visual effects that enhance the visual richness of the rendered scene.

5.3 Summary of the System and Areas for Software Modelling Enhancements

5.3.1 DIP and How It Can Be Used for Flexible Rendering

The Dependency Inversion Principle (DIP) is a key consideration for future enhancements, promoting high flexibility within the rendering pipeline. By

decoupling high-level components from low-level implementations, DIP allows for more adaptable code that can easily incorporate new algorithms or hardware optimizations. This approach not only facilitates maintenance and testing but also ensures that the system can quickly adapt to evolving industry standards and technological advancements.

6 Conclusion

6.1 Summary of Key Points

Monte Carlo methods are effective for path tracing and realistic image synthesis. Importance sampling and stratified sampling significantly improve image quality and convergence speed. Variance reduction techniques further enhance the rendering efficiency by reducing noise.

6.2 Future Work

Future work could explore more advanced sampling strategies, real-time rendering optimizations, and additional variance reduction techniques.

6.3 Final Thoughts

The implementation of these techniques in a practical renderer highlights the importance of statistical methods in achieving high-quality, efficient rendering solutions.

7 References

7.1 Books and Articles

7.2 Online Resources

A Detailed Mathematical Derivations

B Code Snippets and Pseudocode

C Additional Figures and Diagrams