UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia

FAKULTAS ILMU KOMPUTER

CSCE604029 • Computer Graphics
Semester Gasal 2025/2026
Fakultas Ilmu Komputer, Universitas Indonesia

Assignment 3: Ray Tracing

Deadline: 23 November 2025 pukul 23.59 WIB

Alignment with CPMK

- Menguasai konsep-konsep dasar grafika komputer secara teoritis seperti model kamera, properti warna dan cahaya, rendering, iluminasi, dan geometri.
- Dapat mendesain solusi grafika komputer untuk pembentukan citra digital menggunakan teknik pemodelan dasar, terutama menggunakan metode berbasis rasterization dan ray tracing.

1 Introduction

Path tracing represents one of the most elegant and physically accurate rendering techniques in computer graphics, capable of producing photorealistic images by simulating the actual behavior of light. Unlike traditional rasterization techniques that rely on approximations and tricks to achieve realistic lighting, path tracing follows the physical laws of light transport, tracing rays of light as they bounce through a scene and interact with various materials.

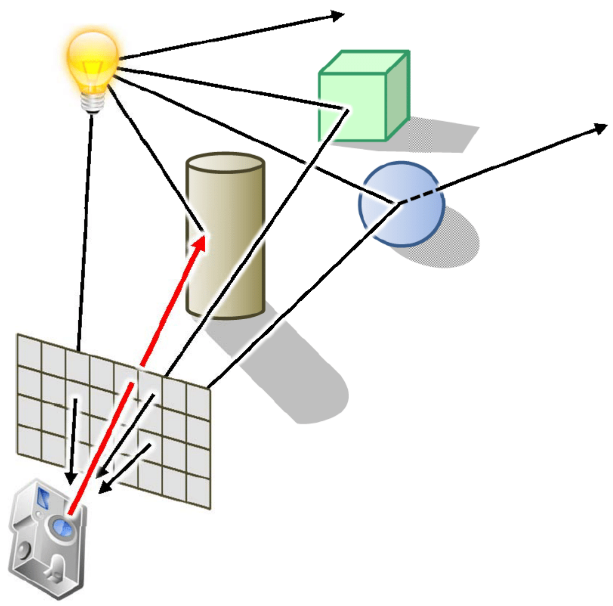
In this assignment, you will implement a complete path tracer from scratch, gaining hands-on experience with the fundamental algorithms that power modern production renderers used in films and visual effects. This journey will take you through the mathematical foundations of light transport, the practical algorithms for ray-object intersection, the physics of material interaction, and the statistical methods needed to solve the rendering equation efficiently.

The path tracer you will build operates on a simple yet powerful principle: to determine the color of a pixel, we trace a ray from the camera through that pixel into the scene, then follow that ray as it bounces off surfaces, accumulating color and light along the way. This Monte Carlo approach, while computationally intensive, provides an unbiased solution to the rendering equation, meaning that with enough samples, it will converge to the physically correct result. Through this implementation, you will understand why path tracing has become the gold standard for photorealistic rendering in the film industry and is increasingly being adopted for real-time applications with the advent of ray tracing hardware.

1.1 Ray Tracing Fundamentals

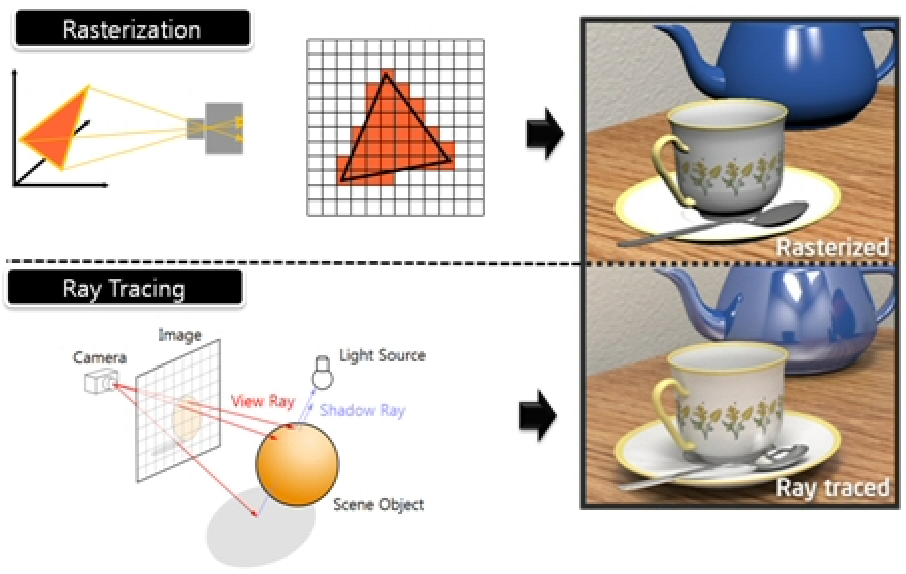
1.1.1 What is Ray Tracing?

Ray tracing is a rendering technique that simulates the physical behavior of light to generate images with unprecedented realism and accuracy. Unlike rasterization, which projects 3D geometry onto a 2D screen using a series of linear transformations and then applies shading models to approximate lighting, ray tracing works by tracing the path of light rays as they travel through a virtual scene. This fundamental difference allows ray tracing to naturally handle complex optical phenomena such as reflections, refractions, shadows, and indirect illumination that are challenging or impossible to achieve accurately with rasterization.



The conceptual foundation of ray tracing is elegantly simple: we simulate light transport by following rays of light backwards from the camera into the scene. For each pixel in the output image, we cast a ray from the camera through that pixel and into the 3D world. When this ray intersects with an object, we calculate how light interacts with that surface, potentially spawning new rays for reflections, refractions, or shadow testing. This process continues recursively until we've gathered enough information to determine the final color of the pixel.

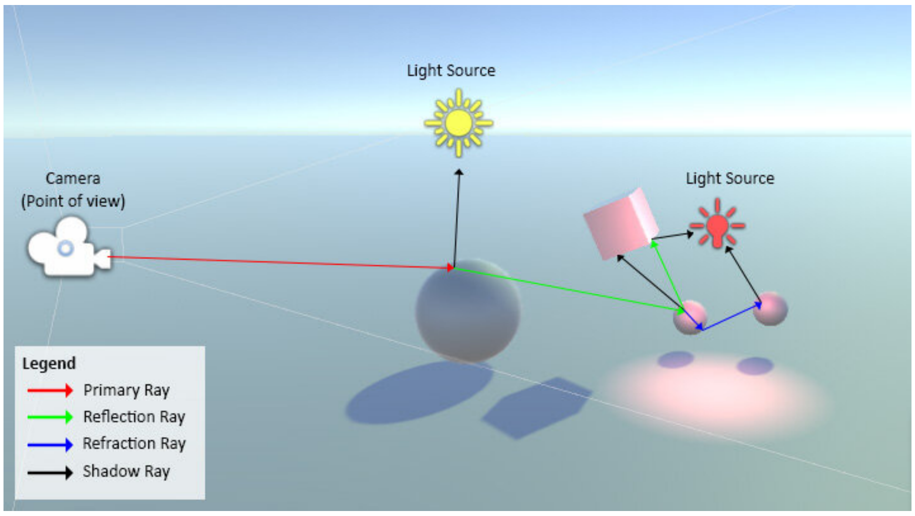
The distinction between ray tracing and rasterization becomes particularly apparent when considering visibility and shading. In rasterization, visibility is determined through the z-buffer algorithm, where fragments compete for each pixel based on their depth values. This approach is inherently local—each primitive is processed independently without knowledge of the rest of the scene. Consequently, effects like shadows require additional rendering passes with shadow maps, reflections need environment mapping or screen-space techniques, and global illumination requires complex approximations. Ray tracing, by contrast, has global knowledge of the scene at every intersection point. When a ray hits a surface, we can cast additional rays to any other part of the scene to gather information about shadows, reflections, or indirect lighting. This global approach makes ray tracing conceptually simpler for complex lighting effects, though computationally more expensive.



The camera model in ray tracing typically uses a pinhole camera abstraction, where all rays originate from a single point (the camera position) and pass through a virtual image plane positioned between the camera and the scene. Each pixel on this image plane corresponds to a specific direction in which we cast a ray. The field of view determines the relationship between pixel positions and ray directions, with wider fields of view spreading rays over larger angles. This model naturally handles perspective projection without requiring the matrix transformations used in rasterization pipelines.

Ray tracing distinguishes between several types of rays, each serving a specific purpose in the light transport simulation:

- **Primary rays** (or camera rays) are the initial rays cast from the camera through each pixel. These rays determine the first visible surface for each pixel and form the foundation of the rendering process.
- **Secondary rays** are spawned from surface intersection points to compute additional light transport contributions. In classical Whitted-style ray tracing, these typically include *reflection rays* (following the mirror reflection direction) and *refraction rays* (passing through transparent or refractive materials). In path tracing, secondary rays also include *diffuse rays* that are sampled over the hemisphere above the surface to account for global illumination and indirect light bounces.
- **Shadow rays** (or occlusion rays) are cast from a surface point toward light sources to determine visibility. If a shadow ray reaches the light without hitting any other geometry, the surface point receives direct illumination from that light; otherwise, it remains in shadow.



The power of ray tracing lies in its ability to accurately model light transport, but this accuracy comes at a computational cost. Each ray-scene intersection requires testing against potentially thousands or millions of geometric primitives, making naive implementations impractically slow. This is why acceleration structures like Bounding Volume Hierarchies (BVH) are essential for practical ray tracing, reducing the average-case complexity from O(n) to O(log n) for n primitives.

1.1.2 Ray Equation and Representation

The mathematical representation of a ray forms the foundation of all ray tracing algorithms. A ray is fundamentally a half-line that starts at a specific point and extends infinitely in a particular direction. We represent this mathematically using the parametric equation:

$$\mathbf{P}(t) = \mathbf{O} + t\mathbf{d} \tag{1}$$

where $\mathbf{O}$ is the ray origin (a 3D point), $\mathbf{d}$ is the ray direction (a 3D vector, typically normalized), and t is a scalar parameter that controls how far along the ray we are. When $t = 0$, we're at the ray origin; as t increases, we move further along the ray in the direction $\mathbf{d}$.

This parametric representation is particularly powerful because it allows us to express any point along the ray using a single scalar value t. When we perform ray-object intersection tests, we're essentially solving for the values of t where the ray equation equals points on the object's surface. The parameter t thus serves multiple purposes: it tells us whether an intersection exists (real, positive values of t), where the intersection occurs (the 3D point $\mathbf{P}(t)$), and which intersection is closest to the ray origin (the smallest positive t).

In practical implementations, we often need to restrict the valid range of t values. For example, we might only want intersections between t_{min} and t_{max}, where $t_{min} > 0$ prevents self-intersections due to numerical errors, and t_{max} limits the ray to a specific distance. This range restriction is crucial for shadow rays, where we only care about intersections between the surface and the light source, not beyond.

The ray structure in code typically contains more than just the origin and direction. A practical ray representation might include:

```
typedef struct {
    vec3 origin;      // Starting point of the ray
    vec3 direction;   // Direction vector (usually normalized)
    float t_min;      // Minimum valid t value
    float t_max;      // Maximum valid t value
    float time;       // Time value for motion blur
    int depth;          // Recursion depth for ray tree
} Ray;
```

The direction vector is usually normalized (unit length) for computational convenience, though this is not strictly necessary. Normalized directions simplify many calculations, such as computing the cosine of angles using dot products. However, some algorithms may use non-normalized directions for specific purposes, such as differentials for texture filtering or importance sampling.

Understanding the ray parameter t is crucial for implementing correct intersection algorithms. The value of t represents the distance along the ray when the direction is normalized, making it directly usable for depth comparisons and distance calculations. When testing multiple objects for intersection, we typically maintain the closest intersection found so far and update it only when we find a closer one (smaller positive t). This nearest-intersection logic is fundamental to determining visibility in ray tracing.

1.1.3 The Rendering Equation

The rendering equation, formulated by James Kajiya in 1986, provides the mathematical foundation for all physically-based rendering techniques, including path tracing. This integral equation describes the equilibrium of light energy in a scene and encompasses all possible light interactions including direct illumination, indirect illumination, reflections, refractions, and every other lighting phenomenon we observe in the real world.

The rendering equation is expressed as:

$$L_o(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + \int_\Omega f_r(\mathbf{x}, \omega_i, \omega_o) L_i(\mathbf{x}, \omega_i)(\omega_i \cdot \mathbf{n})\, d\omega_i \tag{2}$$

Let's break down each component of this equation to understand its physical meaning:

- $L_o(\mathbf{x}, \omega_o)$ represents the outgoing radiance from point $\mathbf{x}$ in direction ω_o. This is what we're trying to compute—the amount of light leaving a surface point toward the camera or another surface.
- $L_e(\mathbf{x}, \omega_o)$ is the emitted radiance from point $\mathbf{x}$ in direction ω_o. This term is non-zero only for light sources and represents the light that the surface generates itself rather than reflecting from other sources.
- $\int_\Omega$ denotes integration over the hemisphere Ω above the surface point. This integral captures the contribution of light arriving from all possible directions in the hemisphere.
- $f_r(\mathbf{x}, \omega_i, \omega_o)$ is the Bidirectional Reflectance Distribution Function (BRDF), which describes how light is reflected at the surface. It tells us what fraction of light coming from direction ω_i is reflected toward direction ω_o.
- $L_i(\mathbf{x}, \omega_i)$ is the incoming radiance at point $\mathbf{x}$ from direction ω_i. This is the light arriving at the surface from other surfaces or light sources.
- $(\omega_i \cdot \mathbf{n})$ is the cosine term, representing the dot product between the incoming light direction and the surface normal. This accounts for the geometric foreshortening effect—light arriving at grazing angles illuminates a larger surface area and thus contributes less radiance per unit area.

The profound insight of the rendering equation is that it's recursive: the incoming radiance L_i at one point is the outgoing radiance L_o from another point in the scene. This recursion captures the infinite bounces of light that create complex illumination effects like color bleeding and caustics. When red light bounces off a red wall onto a white ceiling, turning it slightly pink, this is the rendering equation's recursive nature at work.

The challenge in solving the rendering equation lies in its recursive integral form. The incoming radiance $L_i(\mathbf{x}, \omega_i)$ at a point depends on the outgoing radiance from all other visible points in the scene, which in turn depends on their incoming radiance, creating an infinite system of interdependent equations. Analytical solutions exist only for extremely simple scenes, making numerical methods essential for practical rendering.

This is where Monte Carlo integration becomes invaluable. Instead of trying to evaluate the integral exactly, we approximate it by randomly sampling directions in the hemisphere and averaging their contributions. With enough samples, this stochastic approach converges to the correct solution, though the convergence rate is relatively slow ($O(1/\sqrt{n})$ for n samples), which is why path tracing requires many samples per pixel to produce noise-free images.

The rendering equation also reveals why certain lighting effects are challenging to compute efficiently. Direct illumination (light that travels directly from a light source to a surface to the camera) involves only one bounce and is relatively straightforward to compute. However, indirect illumination involves multiple bounces, requiring recursive evaluation of the rendering equation. Each additional bounce adds another level of integration, exponentially increasing the computational complexity. This is why global illumination has historically been so expensive to compute and why approximations and caching schemes have been developed for real-time applications.

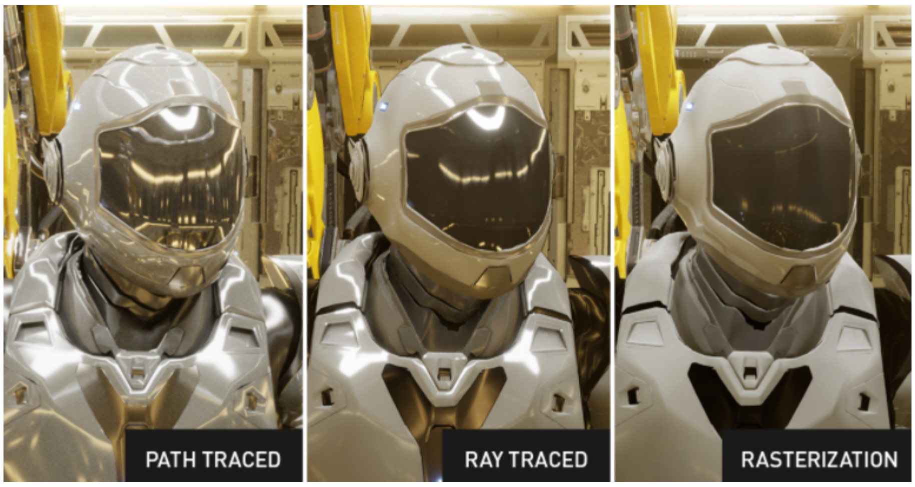
Understanding the rendering equation is crucial for implementing a path tracer because it provides the theoretical foundation for every implementation decision. The way we sample rays, the importance sampling strategies we employ, the Russian roulette termination criteria we use—all of these techniques are motivated by the need to efficiently evaluate this fundamental equation. When you implement your path tracer, you're essentially building a Monte Carlo estimator for the rendering equation, transforming abstract mathematical concepts into concrete algorithms that produce beautiful, physically accurate images.

1.2 Path Tracing Theory

1.2.1 From Ray Tracing to Path Tracing

The evolution from classical ray tracing to path tracing represents a fundamental shift in how we approach the problem of rendering photorealistic images. While Whitted-style ray tracing, introduced in 1980, was revolutionary for its time in generating images with reflections and refractions, it fundamentally operates on an ad-hoc model of light transport. In Whitted ray tracing, we explicitly trace rays for specific phenomena: one ray for reflection on shiny surfaces, one for refraction through transparent materials, and shadow rays to determine direct illumination. This approach, while intuitive and efficient for certain effects, fails to capture the full complexity of light transport in the real world.

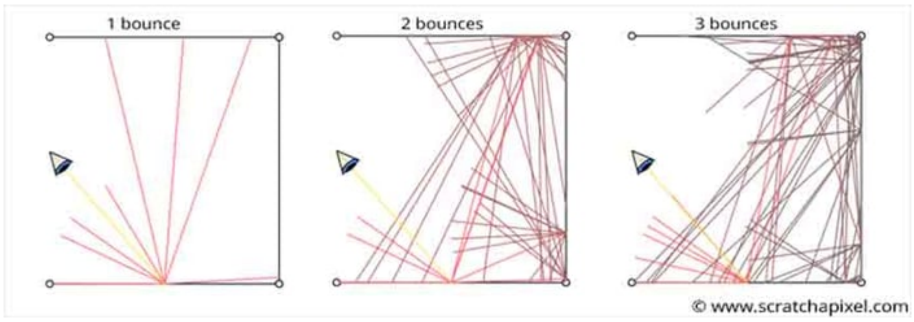
The limitations of classical ray tracing become apparent when we consider diffuse interreflection, commonly known as color bleeding. When light bounces off a red wall onto a white ceiling, the ceiling should appear slightly pink—this is indirect illumination at work. Whitted ray tracing cannot capture this effect because it only traces specular rays and direct illumination. Similarly, caustics (the bright patterns created when light focuses through refractive objects), soft shadows from area lights, and the subtle gradations of ambient occlusion all require a more comprehensive approach to light transport.



Path tracing emerged as the solution to these limitations by taking a fundamentally different approach: instead of tracing specific types of rays for specific phenomena, it traces paths of light as they would physically travel through the scene. Each path represents one possible route that light could take from a light source to the camera, and by tracing many such paths and averaging their contributions, we converge on the correct solution to the rendering equation. This shift from deterministic, phenomenon-specific ray tracing to stochastic, physically-based path tracing enables the simulation of all light transport phenomena within a single, unified framework.

The key insight that makes path tracing practical is the principle of reciprocity in light transport. Light traveling from point A to point B follows the same path as light traveling from B to A, just in reverse. This means we can trace paths backward from the camera into the scene, which is far more efficient than tracing paths forward from light sources, as the vast majority of light emitted by sources never reaches the camera. This backward path tracing approach ensures that every path we trace contributes to the final image, making the computation tractable despite the infinite dimensional integral we're trying to solve.

The distinction between local and global illumination further illustrates the power of path tracing. Local illumination models, including those used in Whitted ray tracing and rasterization, only consider light that travels directly from light sources to surfaces. They might add ambient terms or environment mapping to approximate indirect lighting, but these are hacks that don't accurately model the physics. Path tracing, by contrast, naturally handles global illumination because each ray bounce can gather light from any source, whether direct or indirect. When a path bounces multiple times, it accumulates the contribution of light that has itself bounced multiple times, automatically capturing all orders of indirect illumination.



1.2.2 Recursive Ray Tracing Algorithm

The recursive formulation of path tracing elegantly mirrors the recursive nature of the rendering equation itself. At its core, the algorithm is surprisingly simple: determine the color seen along a ray by finding what it hits, determining how light scatters at that surface, and recursively tracing a new ray in the scattered direction. This directly implements the physics of light transport without special cases or ad-hoc approximations.

Basic Algorithm Structure:

```
function trace_ray(ray, depth):
    if depth > max_depth:
        return black

    hit = find_closest_intersection(ray, scene)
    if no hit:
        return environment_color(ray.direction)

    if hit.material.is_emissive:
        return hit.material.emission

    scattered_ray = sample_bsdf(hit.material, ray, hit)
    if scattered_ray is absorbed:
        return black

    incoming_light = trace_ray(scattered_ray, depth + 1)
    return hit.material.albedo * incoming_light * cosine_term
```

This deceptively simple algorithm encapsulates the entire light transport simulation. Each recursive call represents one bounce along a light path, with the multiplication of colors implementing the throughput calculation as recursion unwinds.

The depth parameter prevents infinite recursion in closed environments. Without proper termination:

- Paths could bounce forever in closed scenes, causing stack overflow
- Even with absorption, expected path lengths might be extremely long
- Computation wastes on paths contributing negligibly to the final image

Maximum depth is typically set between 5 and 50 depending on scene complexity. However, hard cutoffs introduce bias by systematically ignoring longer paths, causing darkening in scenes with multiple indirect bounces.

Russian roulette provides an unbiased alternative to fixed depth cutoffs:

- Randomly terminate paths with probability based on their throughput
- Continue paths with probability $p = \min(1, \text{throughput})$
- Scale surviving paths by $1/p$ to maintain correct expected value
- Efficiently terminates low-contribution paths while preserving unbiased results

This ensures the estimator remains mathematically correct while avoiding wasted computation on negligible paths.

Color accumulation along paths requires careful throughput tracking:

- Throughput = product of all BRDFs and cosine terms along the path
- Each surface interaction attenuates light based on material properties
- A diffuse surface might reflect 80% of incident light

- Geometric terms further reduce by cosine of incident angle
- All factors multiply together maintaining energy conservation

Incorrect throughput calculation leads to implausibly bright images (energy amplification) or unrealistically dark renders (excessive attenuation).

1.3 Ray-Primitive Intersection

Ray-primitive intersection forms the computational core of any ray tracer, consuming the majority of rendering time as millions or billions of rays must be tested against geometric primitives in the scene. The efficiency and correctness of these intersection routines directly determine both the performance and visual quality of the final render. While conceptually straightforward—we're simply finding where a line intersects with geometric shapes—the implementation requires careful attention to numerical precision, edge cases, and computational efficiency. These intersection algorithms must be robust enough to handle degenerate cases, fast enough to execute billions of times per image, and precise enough to avoid visual artifacts from numerical errors.

1.3.1 Ray-Sphere Intersection

The ray-sphere intersection algorithm serves as the perfect introduction to ray-primitive intersection, combining mathematical elegance with practical importance. Spheres appear frequently in ray tracing contexts, from test scenes that validate renderer correctness to production scenes where they approximate particles, droplets, or distant objects. The intersection algorithm derives from the fundamental definition of a sphere—all points equidistant from a center—combined with the parametric ray equation, yielding a quadratic equation whose solutions correspond to intersection points.

The mathematical derivation begins with the sphere equation and ray equation. A sphere with center $\mathbf{C}$ and radius r satisfies the equation $|\mathbf{P} - \mathbf{C}|^2 = r^2$ for any point $\mathbf{P}$ on its surface. The ray equation gives us $\mathbf{P}(t) = \mathbf{O} + t\mathbf{d}$, where $\mathbf{O}$ is the ray origin and $\mathbf{d}$ is the direction. Substituting the ray equation into the sphere equation, we obtain:

$$|\mathbf{O} + t\mathbf{d} - \mathbf{C}|^2 = r^2 \tag{3}$$

Expanding this equation yields a standard quadratic in t:

$$(\mathbf{d} \cdot \mathbf{d})t^2 + 2(\mathbf{d} \cdot \mathbf{oc})t + (\mathbf{oc} \cdot \mathbf{oc} - r^2) = 0 \tag{4}$$

where $\mathbf{oc} = \mathbf{O} - \mathbf{C}$ is the vector from the sphere center to the ray origin.

This quadratic formulation reveals the three possible intersection scenarios through the discriminant. When the discriminant is negative, the ray misses the sphere entirely—there are no real solutions for t. A zero discriminant indicates that the ray grazes the sphere tangentially, touching at exactly one point. A positive discriminant means the ray passes through the sphere, entering at one point and exiting at another, corresponding to two distinct t values.

The standard quadratic formula provides the solutions:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{5}$$

However, a crucial optimization recognizes that if the ray direction is normalized (which is common but not required), then $a = \mathbf{d} \cdot \mathbf{d} = 1$. Furthermore, we can use the half-b formulation where $\text{half}_b = \mathbf{d} \cdot \mathbf{oc}$, simplifying our equation and reducing computational cost. This optimization matters when performing billions of intersection tests.

The interpretation of the t values requires careful consideration. The two solutions represent the entry and exit points of the ray through the sphere. For primary rays from the camera, we typically want the closest intersection, corresponding to the smaller positive t value. However, negative t values represent intersections behind the ray origin, which should be ignored for camera rays but might be relevant for other ray types. The case where one t is negative and one is positive indicates that the ray origin is inside the sphere—a situation that requires special handling depending on the application.

Normal vector calculation at the intersection point is straightforward for spheres. The normal at any point on a sphere points directly away from the center, making it simply the normalized vector from center to intersection point:

$$\mathbf{n} = \frac{\mathbf{P}(t) - \mathbf{C}}{r} \tag{6}$$

The division by radius simultaneously normalizes the vector and saves a square root operation compared to explicit normalization.

Numerical precision issues plague ray-sphere intersection despite its mathematical simplicity. When the ray origin is very close to the sphere surface (common for secondary rays), floating-point errors in the discriminant calculation can cause self-intersection artifacts. The standard solution involves offsetting the minimum valid t value slightly above zero (typically $t_{\text{min}} = 0.001$), preventing rays from immediately re-intersecting the surface they just left. This epsilon value must be chosen carefully—too small and self-intersections persist, too large and we create visible gaps between objects.

1.3.2 Ray-Triangle Intersection

Triangle intersection represents the most critical primitive intersection in ray tracing, as triangulated meshes form the basis of most complex geometry in computer graphics. Every detailed model, from characters to environments, ultimately decomposes into triangles for rendering. The efficiency of ray-triangle intersection therefore directly impacts the performance of production ray tracers. The Möller-Trumbore algorithm has emerged as the standard solution, providing an elegant and efficient method that simultaneously computes the intersection point and barycentric coordinates without requiring pre-computed plane equations or coordinate system transformations.

The Möller-Trumbore algorithm leverages the parametric representation of a point within a triangle using barycentric coordinates. Any point $\mathbf{P}$ inside a triangle with vertices $\mathbf{V_0}$, $\mathbf{V_1}$, and $\mathbf{V_2}$ can be expressed as:

$$\mathbf{P} = (1 - u - v)\mathbf{V_0} + u\mathbf{V_1} + v\mathbf{V_2} \tag{7}$$

where u and v are the barycentric coordinates, satisfying $u \geq 0$, $v \geq 0$, and $u + v \leq 1$ for points inside the triangle. This can be rewritten as:

$$\mathbf{P} = \mathbf{V_0} + u(\mathbf{V_1} - \mathbf{V_0}) + v(\mathbf{V_2} - \mathbf{V_0}) \tag{8}$$

Setting this equal to the ray equation $\mathbf{O} + t\mathbf{d}$ and rearranging gives us a system of linear equations:

$$\mathbf{O} - \mathbf{V_0} = -t\mathbf{d} + u\mathbf{E_1} + v\mathbf{E_2} \tag{9}$$

where $\mathbf{E_1} = \mathbf{V_1} - \mathbf{V_0}$ and $\mathbf{E_2} = \mathbf{V_2} - \mathbf{V_0}$ are the edge vectors.

The brilliance of the Möller-Trumbore algorithm lies in solving this system using Cramer's rule, which expresses the solution in terms of determinants that can be computed efficiently using cross and dot products. The solution involves computing:

$$\mathbf{h} = \mathbf{d} \times \mathbf{E_2} \tag{10}$$

$$a = \mathbf{E_1} \cdot \mathbf{h} \tag{11}$$

$$\mathbf{s} = \mathbf{O} - \mathbf{V_0} \tag{12}$$

$$u = \frac{1}{a}(\mathbf{s} \cdot \mathbf{h}) \tag{13}$$

If $|a|$ is very small (close to zero), the ray is nearly parallel to the triangle plane, and we can immediately reject the intersection. This early rejection test is one of the algorithm's strengths, avoiding unnecessary computation for rays that cannot intersect the triangle.

The algorithm continues with similar calculations for v and t:

$$\mathbf{q} = \mathbf{s} \times \mathbf{E_1} \tag{14}$$

$$v = \frac{1}{a}(\mathbf{d} \cdot \mathbf{q}) \tag{15}$$

$$t = \frac{1}{a}(\mathbf{E_2} \cdot \mathbf{q}) \tag{16}$$

The beauty of this formulation is that we can test the barycentric coordinates as we compute them, enabling early rejection without completing all calculations. If $u < 0$ or $u > 1$, we can immediately return no intersection. Similarly, if $v < 0$ or $u + v > 1$, the point lies outside the triangle. Only if all barycentric conditions are satisfied do we need to check if t falls within the valid range $[t_{\min}, t_{\max}]$.

Normal calculation for triangles requires careful consideration of winding order and double-sided surfaces. The geometric normal can be computed from the cross product of edge vectors:

$$\mathbf{n} = \frac{\mathbf{E_1} \times \mathbf{E_2}}{|\mathbf{E_1} \times \mathbf{E_2}|} \tag{17}$$

However, this normal points in a direction determined by the vertex winding order. For single-sided triangles (common in closed meshes), we might cull back-facing triangles by checking if $\mathbf{d} \cdot \mathbf{n} > 0$. For double-sided triangles (used in thin surfaces like leaves or cloth), we need to flip the normal for back-face hits to ensure it always points toward the ray origin.

The algorithm's efficiency stems from its minimal arithmetic operations and early rejection opportunities. Compared to the classical approach of computing the plane equation, transforming to 2D, and testing point-in-triangle, Möller-Trumbore requires fewer operations and no coordinate system changes. The algorithm performs one cross product, five dot products, one reciprocal, three multiplies, and three adds in the worst case, with opportunities for early termination reducing the average cost further.

1.3.3 Ray-AABB Intersection

Axis-Aligned Bounding Box (AABB) intersection serves as the gateway operation for hierarchical acceleration structures, determining which parts of the BVH tree need detailed intersection testing. Unlike sphere or triangle intersections that directly contribute to the rendered image, AABB intersections act as conservative filters, quickly rejecting large portions of the scene that cannot possibly intersect with a ray. The efficiency of AABB intersection is therefore paramount—it must be as fast as possible while maintaining conservative correctness, never producing false negatives that would cause us to miss actual geometry intersections.

The conceptual foundation of AABB intersection rests on the slab method, which views an AABB as the intersection of three pairs of parallel planes aligned with the coordinate axes. Each pair of planes forms an infinite slab, and the AABB is the region where all three slabs overlap. A ray intersects the AABB if and only if there exists a segment of the ray that lies within all three slabs simultaneously. This geometric insight transforms the 3D intersection problem into three independent 1D problems, which can be solved efficiently and combined to determine the overall intersection.

For each coordinate axis, we compute where the ray enters and exits the corresponding slab. Given an AABB with minimum corner $\mathbf{min} = (x_{\text{min}}, y_{\text{min}}, z_{\text{min}})$ and maximum corner $\mathbf{max} = (x_{\text{max}}, y_{\text{max}}, z_{\text{max}})$, and a ray with origin $\mathbf{O} = (O_x, O_y, O_z)$ and direction $\mathbf{d} = (d_x, d_y, d_z)$, the t values for entering and exiting the x-slab are:

$$t_{x,\text{min}} = \frac{x_{\text{min}} - O_x}{d_x} \tag{18}$$

$$t_{x,\text{max}} = \frac{x_{\text{max}} - O_x}{d_x} \tag{19}$$

Similar calculations apply for the y and z slabs. However, we must be careful about the sign of the direction components. If d_x is negative, the ray travels in the negative x direction, so it enters the slab at x_{max} and exits at x_{min}, swapping our t values. Rather than using conditional logic, we can handle this elegantly by always computing both t values and then using minimum and maximum operations to determine entry and exit.

The overall intersection logic combines the per-axis results. The ray enters the AABB at the latest entry point across all axes and exits at the earliest exit point:

$$t_{\text{enter}} = \max(t_{x,\text{min}}, t_{y,\text{min}}, t_{z,\text{min}}) \tag{20}$$

$$t_{\text{exit}} = \min(t_{x,\text{max}}, t_{y,\text{max}}, t_{z,\text{max}}) \tag{21}$$

An intersection occurs if and only if $t_{\text{enter}} \leq t_{\text{exit}}$ and the intersection interval $[t_{\text{enter}}, t_{\text{exit}}]$ overlaps with the ray's valid range $[t_{\text{min}}, t_{\text{max}}]$.

Special cases require careful handling to maintain robustness. When a ray direction component is zero or very small, the division can produce infinity or large numerical errors. For exactly zero components, we can use a separate test: if the ray origin lies outside the slab in that dimension, there's no intersection; otherwise, the ray is parallel to and inside the slab, contributing $[-\infty, +\infty]$ to the intersection interval. For near-zero components, we might use a small epsilon to avoid division by very small numbers, though modern implementations often rely on IEEE floating-point infinity handling to manage these cases naturally.

1.3.4 Hit Record Structure

The hit record serves as the communication protocol between intersection routines and shading code, encapsulating all information needed to shade an intersection point. This abstraction allows the shading system to remain agnostic about primitive types while providing all necessary data for rendering calculations.

The practical C structure contains:

```
typedef struct {
    float t;                 // Ray parameter at intersection
    vec3 point;             // 3D intersection point
    vec3 normal;            // Surface normal (toward ray)
    vec3 geometric_normal;  // True geometric normal
    vec2 uv;                // Texture coordinates
    Material* material;     // Material pointer
    bool front_face;        // Hit front or back face
    int primitive_id;       // For debugging/AOVs
} HitRecord;
```

Each field serves a specific purpose:

- `t`: Determines visibility and computes the 3D intersection point
- `point`: Cached intersection position for repeated use during shading
- `normal`: Surface normal oriented toward ray origin for shading calculations
- `geometric_normal`: Unmodified normal for special effects and transparency handling
- `uv`: Texture coordinates for mapping 2D textures to 3D surfaces
- `material`: Pointer to material properties (keeps hit record compact)
- `front_face`: Indicates which side of the surface was hit
- `primitive_id`: Useful for debugging and render passes

Usage Lifecycle:
The hit record flows through two main phases:

- *Intersection phase*: Geometric data (t, point, normal, UV) is computed and stored
- *Shading phase*: Data is read repeatedly for material evaluation and light calculations

Since millions of hit records are created per frame:

- Stack allocation preferred over heap for performance
- Small structure size improves cache utilization
- Field ordering affects cache line alignment
- Material stored as pointer to minimize record size

This abstraction provides clean separation between geometry and shading, enabling intersection routines to be reused across different rendering algorithms while maintaining efficient memory usage and performance.

1.4 Materials and BRDF

The appearance of objects in the physical world emerges from the complex interaction between light and matter at the microscopic scale. When light strikes a surface, photons interact with the material's atomic structure through absorption, reflection, and transmission processes that depend on the material's chemical composition, surface structure, and electromagnetic properties. In computer graphics, we abstract these intricate physical phenomena into mathematical models called Bidirectional Reflectance Distribution Functions (BRDFs) that capture the essential characteristics of how materials scatter light while remaining computationally tractable for rendering algorithms.

1.4.1 Bidirectional Reflectance Distribution Function (BRDF)

The Bidirectional Reflectance Distribution Function stands as one of the fundamental concepts in physically-based rendering, providing a mathematical framework for describing how light reflects off opaque surfaces. The BRDF encodes the relationship between incoming and outgoing light at a surface point, answering the essential question: given light arriving from a particular direction, how much of that light is reflected toward another specific direction? This function forms the cornerstone of the rendering equation, determining the appearance of every non-emissive surface in our scenes.

Formally, the BRDF $f_r(\omega_i, \omega_o)$ is defined as the ratio of differential radiance reflected in direction ω_o to the differential irradiance arriving from direction ω_i:

$$f_r(\omega_i, \omega_o) = \frac{dL_o(\omega_o)}{dE_i(\omega_i)} = \frac{dL_o(\omega_o)}{L_i(\omega_i)\cos\theta_i \, d\omega_i} \tag{22}$$

where L_o is the outgoing radiance, L_i is the incoming radiance, θ_i is the angle between the incoming direction and the surface normal, and $d\omega_i$ is the differential solid angle. The units of BRDF are inverse steradians (sr^{-1}), representing a distribution rather than a simple ratio.

For a BRDF to be physically plausible, it must satisfy several fundamental properties rooted in physics. The first and most critical is energy conservation: a surface cannot reflect more light than it receives. Mathematically, this constraint requires that the integral of the BRDF over all outgoing directions, weighted by the cosine term, must not exceed unity:

$$\int_\Omega f_r(\omega_i, \omega_o)\cos\theta_o \, d\omega_o \leq 1 \tag{23}$$

This inequality ensures that our materials don't artificially brighten the scene by creating energy from nothing, a common artifact in early computer graphics when ad-hoc shading models violated energy conservation.

The second fundamental property is Helmholtz reciprocity, which states that the BRDF must be symmetric with respect to the interchange of incoming and outgoing directions:

$$f_r(\omega_i, \omega_o) = f_r(\omega_o, \omega_i) \tag{24}$$

This reciprocity principle emerges from the time-reversibility of electromagnetic radiation and has profound implications for rendering algorithms. It ensures that light paths can be traced equally well in either direction, enabling bidirectional path tracing algorithms and guaranteeing consistent results regardless of whether we trace rays from lights or cameras.

The BRDF must also be non-negative for all direction pairs, as negative reflectance has no physical meaning. While this seems obvious, it becomes a practical concern when implementing approximate or phenomenological BRDF models that might produce negative values due to numerical errors or extrapolation beyond their valid parameter ranges.

1.4.2 Lambertian (Diffuse) Materials

Lambertian materials represent the idealized model of perfectly diffuse reflection, where incident light is scattered equally in all directions regardless of the viewing angle. Named after Johann Heinrich Lambert who described this reflection model in the 18th century, Lambertian surfaces appear equally bright from all viewing directions, a property we observe approximately in materials like chalk, matte paint, and unfinished wood. While no real material is perfectly Lambertian, this model serves as an excellent approximation for many rough surfaces and forms the foundation for more complex material models.

The Lambertian BRDF is remarkably simple, consisting of a constant value that depends only on the surface's albedo (its inherent color or reflectance):

$$f_r^{\text{Lambert}} = \frac{\rho}{\pi} \tag{25}$$

where ρ is the albedo, typically represented as an RGB color with each component in the range [0,1]. The factor of π in the denominator ensures energy conservation. To understand why this specific normalization is needed, consider that when we integrate the Lambertian BRDF over the hemisphere with the required cosine weighting:

$$\int_\Omega \frac{\rho}{\pi}\cos\theta \, d\omega = \rho \int_0^{2\pi} \int_0^{\pi/2} \frac{1}{\pi}\cos\theta \sin\theta \, d\theta \, d\phi = \rho \tag{26}$$

This integration yields exactly ρ, confirming that a white Lambertian surface ($\rho = 1$) reflects all incident light, while colored surfaces reflect only their respective color components.

The physical intuition behind Lambertian reflection relates to surface microstructure. At the microscopic scale, a Lambertian surface consists of countless tiny facets oriented randomly in all directions. When light hits these microfacets, it bounces in random directions determined by the local facet orientation. Since the facets are uniformly distributed over all orientations, the aggregate effect is uniform scattering in all directions. This microscopic chaos averages out to the simple macroscopic behavior described by Lambert's law.

Implementing Lambertian materials in a path tracer requires both evaluation and sampling routines. The evaluation is trivial—simply return ρ/π for any valid direction pair. The sampling strategy, however, deserves careful attention for optimal performance. While we could sample directions uniformly over the hemisphere, this would ignore the cosine term in the rendering equation, leading to high variance. Instead, we use cosine-weighted hemisphere sampling, which generates directions with probability proportional to $\cos\theta$:

$$p(\omega) = \frac{\cos\theta}{\pi} \tag{27}$$

This importance sampling strategy can be implemented elegantly using the Malley's method: generate uniform random points on a unit disk, then project them onto the hemisphere. The resulting directions naturally follow the cosine distribution:

```
vec3 sample_cosine_hemisphere(RNG* rng) {
    float r = sqrt(rng_float(rng));
    float theta = 2 * PI * rng_float(rng);
    float x = r * cos(theta);
    float y = r * sin(theta);
    float z = sqrt(max(0, 1 - x*x - y*y));
    return vec3_create(x, y, z);
}
```

The beauty of cosine-weighted sampling for Lambertian materials lies in the cancellation that occurs in the Monte Carlo estimator. The BRDF value ρ/π, the cosine term $\cos\theta$, and the probability density $\cos\theta/\pi$ combine to yield simply ρ, regardless of the sampled direction. This perfect importance sampling eliminates directional variance, leaving only the variance from different path contributions.

1.4.3 Metallic Materials

Metallic materials exhibit fundamentally different optical properties from dielectrics due to their abundance of free electrons that can respond to electromagnetic fields. When light strikes a metal surface, these mobile electrons oscillate in response to the electric field, re-radiating electromagnetic waves that we perceive as reflection. This electronic response gives metals their characteristic properties: high reflectivity, wavelength-dependent reflection that creates colored metals like gold and copper, and complete opacity even in thin layers. Understanding and modeling these properties accurately is essential for achieving photorealistic rendering of metallic objects.

The ideal metal exhibits perfect specular reflection, where incident light bounces off the surface following the law of reflection: the angle of incidence equals the angle of reflection, with both angles measured relative to the surface normal. For a perfect mirror, the BRDF is a delta function:

$$f_r^{\text{mirror}}(\omega_i, \omega_o) = \frac{F(\omega_i, n)}{\cos\theta_i}\delta(\omega_o - \omega_r) \tag{28}$$

where $\omega_r = 2(\omega_i \cdot n)n - \omega_i$ is the perfect reflection direction, and $F(\omega_i, n)$ is the Fresnel reflectance. The delta function ensures that light is reflected only in the perfect mirror direction, making this BRDF unsuitable for standard Monte Carlo sampling. Instead, we handle perfect mirrors by deterministically generating the reflection ray.

Real metals are never perfectly smooth at the microscopic scale. Surface irregularities, oxidation, and manufacturing processes create microscopic roughness that scatters reflected light into a cone around the perfect reflection direction. This roughness transforms the sharp specular reflection into a broader, softer highlight. We model this phenomenon by introducing a roughness parameter that controls the spread of the reflection lobe. A roughness of zero corresponds to a perfect mirror, while increasing roughness creates increasingly diffuse specular reflection.

The implementation of rough metallic reflection typically uses microfacet theory, which models the surface as a collection of tiny mirror facets with varying orientations. The distribution of facet orientations determines the appearance of the reflection. The most common distribution is the GGX (Trowbridge-Reitz) distribution:

$$D(\omega_h) = \frac{\alpha^2}{\pi((\omega_h \cdot n)^2(\alpha^2 - 1) + 1)^2} \tag{29}$$

where ω_h is the half-vector between the incident and outgoing directions, and α is the roughness parameter. This distribution produces realistic highlights with a bright core and extended tails that match observed materials well.

The complete BRDF for rough metals combines the microfacet distribution with geometric shadowing/masking terms and the Fresnel reflectance:

$$f_r^{\text{metal}} = \frac{D(\omega_h)G(\omega_i, \omega_o)F(\omega_i, \omega_h)}{4\cos\theta_i\cos\theta_o} \tag{30}$$

where G is the geometric term accounting for microfacet self-shadowing. While this complete model provides accurate results, simpler approximations often suffice for basic path tracing.

For path tracing implementation, we can use a simplified approach that captures the essential behavior of metallic reflection without the full complexity of microfacet theory. We generate scattered rays by perturbing the perfect reflection direction with random offsets scaled by the roughness parameter:

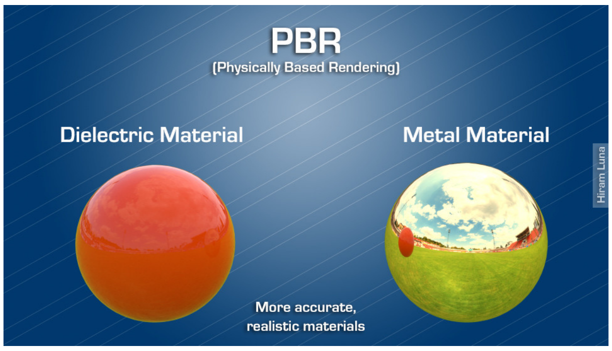
```
vec3 sample_rough_metal(vec3 incident, vec3 normal, float roughness, RNG* rng) {
    vec3 reflected = vec3_reflect(incident, normal);
    vec3 fuzz = vec3_scale(rng_in_unit_sphere(rng), roughness);
    return vec3_normalize(vec3_add(reflected, fuzz));
}
```

This approximation, while not physically accurate, produces visually plausible results and is simple to implement and understand. The roughness parameter directly controls the cone angle of scattered rays, with zero producing perfect reflection and one creating nearly diffuse reflection.

The Fresnel effect for metals is particularly important, as it determines how reflectivity varies with angle. Unlike dielectrics where Fresnel effects are subtle except at grazing angles, metals exhibit strong wavelength-dependent Fresnel behavior even at normal incidence. This wavelength dependence is what gives metals like gold and copper their characteristic colors. For simple path tracers, we often approximate this with a constant base reflectivity (the F0 value) that represents the metal's color, though more sophisticated implementations use the full complex index of refraction to compute accurate Fresnel values.

1.4.4 Dielectric Materials

Dielectric materials—transparent substances like glass, water, and diamond—present unique challenges and opportunities in path tracing due to their ability to both reflect and transmit light. The term "dielectric" originates from their electrical properties as insulators, but in graphics, it refers to any transparent material where light can pass through. The interplay between reflection and refraction at dielectric interfaces creates compelling visual effects: the sparkle of diamonds, the distortion seen through water, and the complex caustic patterns cast by glass objects. Accurately modeling these phenomena requires understanding Snell's law, Fresnel equations, and the careful handling of total internal reflection.


PBR
(Physically Based Rendering)
Dielectric Material
Metal Material
Hiram Luna
More accurate, realistic materials

The fundamental behavior of dielectrics is governed by Snell's law, which describes how light bends when passing between media with different indices of refraction:

$$n_1 \sin\theta_1 = n_2 \sin\theta_2 \tag{31}$$

where n_1 and n_2 are the refractive indices of the two media, and θ_1 and θ_2 are the angles of incidence and refraction relative to the surface normal. The refractive index measures how much light slows down in a medium compared to vacuum, with typical values being 1.0 for air, 1.33 for water, 1.5 for glass, and 2.4 for diamond. This speed change causes light rays to bend at interfaces, creating the magnification and distortion effects we associate with transparent objects.

The direction of the refracted ray can be computed using the vector form of Snell's law:

$$\omega_t = \frac{n_1}{n_2}(\omega_i + \cos\theta_i \cdot n) - n\sqrt{1 - \left(\frac{n_1}{n_2}\right)^2 (1 - \cos^2\theta_i)} \tag{32}$$

This formula assumes the normal points into the medium from which the ray arrives. The term under the square root becomes negative when total internal reflection occurs—a phenomenon unique to rays traveling from a denser to a less dense medium at angles beyond the critical angle.

Total internal reflection occurs when the angle of incidence exceeds the critical angle $\theta_c = \arcsin(n_2/n_1)$ for $n_1 > n_2$. At angles greater than this critical angle, no refraction is possible, and all light is reflected. This phenomenon is responsible for the bright reflections seen at the water surface when viewed from below and enables fiber optic cables to guide light over long distances. In path tracing, we must detect this condition and switch from refraction to reflection when it occurs.

The Fresnel equations determine what fraction of light is reflected versus transmitted at a dielectric interface. For unpolarized light (which we assume in most renderers), the Fresnel reflectance can be computed as:

$$F = \frac{1}{2}\left[\left(\frac{n_1\cos\theta_i - n_2\cos\theta_t}{n_1\cos\theta_i + n_2\cos\theta_t}\right)^2 + \left(\frac{n_1\cos\theta_t - n_2\cos\theta_i}{n_1\cos\theta_t + n_2\cos\theta_i}\right)^2\right] \tag{33}$$

This exact formula is computationally expensive, so Schlick's approximation is commonly used:

$$F \approx F_0 + (1 - F_0)(1 - \cos\theta_i)^5 \tag{34}$$

where $F_0 = ((n_1 - n_2)/(n_1 + n_2))^2$ is the reflectance at normal incidence. This approximation is remarkably accurate for most common materials and significantly faster to compute.

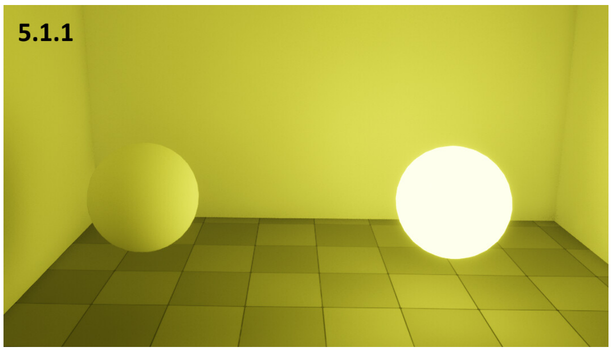
Implementing dielectric materials in a path tracer requires making a discrete choice between reflection and refraction for each ray interaction. Since we can only follow one path in standard path tracing, we randomly choose between reflection and refraction with probabilities based on the Fresnel reflectance:

```
bool sample_dielectric(Ray* ray, HitRecord* hit, vec3* attenuation,
                       Ray* scattered, RNG* rng) {
    float ior = hit->material->ior;
    float ratio = hit->front_face ? (1.0 / ior) : ior;

    vec3 unit_direction = vec3_normalize(ray->direction);
    float cos_theta = fmin(-vec3_dot(unit_direction, hit->normal), 1.0);
    float sin_theta = sqrt(1.0 - cos_theta * cos_theta);

    // Check for total internal reflection
    bool cannot_refract = ratio * sin_theta > 1.0;
    vec3 direction;

    if (cannot_refract || schlick(cos_theta, ratio) > rng_float(rng)) {
        direction = vec3_reflect(unit_direction, hit->normal);
    } else {
        direction = vec3_refract(unit_direction, hit->normal, ratio);
    }

    *scattered = ray_create(hit->point, direction);
    *attenuation = vec3_create(1, 1, 1);  // No absorption
    return true;
}
```

This stochastic approach naturally produces the correct ratio of reflected to refracted light over many samples, though it increases variance compared to deterministically tracing both paths and weighting them appropriately.

1.4.5 Emissive Materials

Emissive materials serve as the light sources in path traced scenes, injecting energy into the system that ultimately illuminates all other surfaces through direct and indirect lighting. Unlike traditional computer graphics where lights are separate entities with special handling, path tracing treats emissive materials as regular scene geometry that happens to emit radiance. This unified approach elegantly handles area lights of arbitrary shape, naturally produces soft shadows, and correctly accounts for indirect illumination from glowing surfaces. Every path traced image ultimately derives its illumination from emissive materials, making their proper implementation crucial for both physical accuracy and artistic control.



The emission from a surface is characterized by its emitted radiance L_e, which can vary with position, direction, and wavelength. For the simple case of a uniform diffuse emitter (like a uniformly glowing sphere or plane), the emitted radiance is constant in all directions:

$$L_e(\mathbf{x}, \omega) = E \tag{35}$$

where E is the emission strength, typically specified as an RGB color representing the spectral radiance. More sophisticated emitters might exhibit directional variation, such as spot lights that concentrate emission in particular directions, though these require more complex emission profiles.

The power (flux) emitted by an area light is the integral of radiance over all directions and surface area:

$$\Phi = \int_A \int_\Omega L_e(\mathbf{x}, \omega) \cos\theta \, d\omega \, dA \tag{36}$$

For a diffuse emitter with area A and emission E, this yields $\Phi = \pi AE$. This relationship helps when specifying lights by their total power output rather than radiance, ensuring physically meaningful brightness relationships between lights of different sizes.

In path tracing implementation, emissive materials require special handling to avoid infinite recursion and ensure energy conservation. When a ray hits an emissive surface, we don't scatter new rays from that point—emission is a source, not a scattering event. The path tracing algorithm must detect emissive materials and add their contribution to the accumulated radiance:

```
if (hit.material->type == MATERIAL_EMISSIVE) {
    // Only emit light on the front face (optional)
    if (hit.front_face) {
        return hit.material->emission;
    }
    return vec3_create(0, 0, 0);
}
```

This simple approach works but raises important design decisions. Should emissive surfaces emit from both sides or only the front? Should they also reflect light from other sources, or are they purely emissive? These choices affect both the physical plausibility and artistic utility of the lighting system.

The relationship between emissive materials and the rendering equation deserves careful consideration. The rendering equation separates emitted radiance L_e from reflected radiance:

$$L_o(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + \int_\Omega f_r(\mathbf{x}, \omega_i, \omega_o) L_i(\mathbf{x}, \omega_i) \cos\theta_i \, d\omega_i \tag{37}$$

This separation means emissive surfaces contribute to the image in two ways: directly, when visible to the camera, and indirectly, by illuminating other surfaces. The path tracer naturally handles both contributions through its recursive evaluation of the rendering equation.

1.5 Monte Carlo Integration

Monte Carlo integration transforms intractable mathematical problems into statistical estimation tasks, leveraging randomness to solve integrals that would be impossible or impractical to evaluate analytically. Named after the famous casino in Monaco, this method embraces uncertainty and probability to achieve deterministic results, a seemingly paradoxical approach that has revolutionized fields from physics to finance. In computer graphics, Monte Carlo methods enable us to solve the rendering equation—a recursive integral equation with no closed-form solution—by converting the continuous integration problem into a discrete sampling problem that computers can handle efficiently.

1.5.1 The Monte Carlo Method

The Monte Carlo method rests on a profound insight: we can estimate the value of any integral by randomly sampling the integrand and computing the average of these samples. This approach transforms integration from a continuous mathematical operation into a discrete statistical process, making it particularly well-suited for computer implementation. The theoretical foundation lies in the law of large numbers, which guarantees that as we increase the number of samples, our estimate converges to the true value of the integral with probability one.

Consider the problem of evaluating a general integral over some domain Ω:

$$I = \int_\Omega f(x) \, dx \tag{38}$$

The Monte Carlo estimator approximates this integral using N random samples x_i drawn from a probability density function $p(x)$:

$$\langle I \rangle = \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)} \tag{39}$$

This estimator is unbiased, meaning its expected value equals the true integral:

$$E[\langle I \rangle] = E\left[\frac{f(X)}{p(X)}\right] = \int_\Omega \frac{f(x)}{p(x)} p(x) \, dx = \int_\Omega f(x) \, dx = I \tag{40}$$

The power of this formulation becomes apparent when we consider high-dimensional integrals. Traditional numerical integration methods like Simpson's rule or Gaussian quadrature suffer from the curse of dimensionality—their convergence rate deteriorates exponentially with the number of dimensions. Monte Carlo integration, remarkably, maintains a convergence rate of $O(1/\sqrt{N})$ regardless of dimensionality. This dimension-independent convergence makes Monte Carlo the only practical choice for the high-dimensional integrals encountered in light transport, where each bounce of light adds dimensions to the integration domain.

The variance of the Monte Carlo estimator determines the quality of our approximation and the rate of convergence:

$$\text{Var}[\langle I \rangle] = \frac{1}{N}\text{Var}\left[\frac{f(X)}{p(X)}\right] = \frac{1}{N}\left(\int_\Omega \left(\frac{f(x)}{p(x)}\right)^2 p(x)\, dx - I^2\right) \tag{41}$$

This variance decreases linearly with the number of samples, leading to the characteristic $1/\sqrt{N}$ convergence rate for the standard deviation (error). In practical terms, this means that to reduce the error by half, we need four times as many samples—a slow convergence that motivates the development of variance reduction techniques.

The choice of probability density function $p(x)$ profoundly affects the estimator's variance. The ideal choice would be $p(x) \propto |f(x)|$, which would yield zero variance if we knew the normalization constant. Of course, if we knew this normalization, we would already know the integral's value. This observation leads to the fundamental principle of importance sampling: we should sample more densely where the integrand is large and less densely where it's small, approximating the ideal density as closely as possible with the information available.

In the context of rendering, Monte Carlo integration allows us to estimate pixel colors by randomly sampling light paths. Each path represents one sample of the rendering equation's integrand, and the average of many such paths converges to the true pixel color. The randomness manifests as noise in the rendered image—a grainy appearance that gradually smooths out as more samples are added. This noise is the visual representation of variance in our Monte Carlo estimator, and understanding its sources helps us develop better sampling strategies.

1.5.2 Estimating the Rendering Equation

The rendering equation presents a formidable computational challenge: it's a recursive Fredholm integral equation of the second kind with no general analytical solution. Monte Carlo integration provides the key to unlocking this equation, transforming it from an abstract mathematical formulation into a practical algorithm that produces stunning photorealistic images. The application of Monte Carlo to the rendering equation reveals deep connections between probability theory, physics, and visual perception.

Recall the rendering equation in its integral form:

$$L_o(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + \int_\Omega f_r(\mathbf{x}, \omega_i, \omega_o) L_i(\mathbf{x}, \omega_i) \cos\theta_i \, d\omega_i \tag{42}$$

To apply Monte Carlo integration, we need to identify the integrand and choose an appropriate sampling strategy. The integrand is the product of three terms: the BRDF f_r, the incoming radiance L_i, and the cosine term $\cos\theta_i$. The challenge is that L_i itself is unknown—it's the outgoing radiance from another point in the scene, which depends on yet another rendering equation evaluation.

The Monte Carlo estimator for the rendering equation becomes:

$$L_o(\mathbf{x}, \omega_o) \approx L_e(\mathbf{x}, \omega_o) + \frac{1}{N}\sum_{i=1}^{N} \frac{f_r(\mathbf{x}, \omega_i, \omega_o) L_i(\mathbf{x}, \omega_i) \cos\theta_i}{p(\omega_i)} \tag{43}$$

where ω_i are random directions sampled according to the probability density $p(\omega)$.

The recursive nature of the rendering equation leads to the path integral formulation of light transport. When we recursively evaluate L_i using Monte Carlo, we build up a path from the camera into the scene. Each recursive evaluation adds a vertex to the path, and the final path contribution is the product of all BRDFs and cosine terms along the path, multiplied by the emission from the light source at the path's end:

$$L = L_e \prod_{k=1}^{n} \frac{f_r(\mathbf{x}_k, \omega_{k-1}, \omega_k) \cos\theta_k}{p(\omega_k)} \tag{44}$$

This path integral formulation reveals why path tracing can be interpreted as a random walk through the scene. At each surface interaction, we randomly choose a new direction based on the material properties, continuing until we hit a light source or terminate the path. The contribution of each path is weighted by the probability of generating that specific path, ensuring an unbiased estimate of the true radiance.

The choice of sampling density $p(\omega)$ at each bounce significantly impacts the estimator's efficiency. The optimal strategy depends on which factors in the integrand we can importance sample:

- **BRDF Importance Sampling**: Sample directions according to the BRDF's shape, concentrating samples where the BRDF is large. This is particularly effective for specular materials where the BRDF is highly peaked.
- **Cosine Importance Sampling**: Sample directions according to the cosine term, generating more samples near the normal. This works well for diffuse materials where the BRDF is relatively constant.
- **Product Importance Sampling**: Ideally, we would sample according to the product $f_r \cdot \cos\theta$, but this is often difficult to sample directly.
- **Light Importance Sampling**: In advanced implementations, we might sample directions toward bright light sources, though this requires knowledge of the scene's light distribution.

The practical implementation typically uses BRDF importance sampling combined with cosine weighting for diffuse surfaces. For a Lambertian BRDF with cosine-weighted sampling, the Monte Carlo estimator simplifies beautifully:

$$L_o \approx L_e + \frac{1}{N} \sum_{i=1}^{N} \frac{(\rho/\pi) L_i \cos\theta_i}{(\cos\theta_i/\pi)} = L_e + \frac{\rho}{N} \sum_{i=1}^{N} L_i \tag{45}$$

The cosine terms cancel, leaving a simple average of the incoming radiance values multiplied by the albedo. This cancellation is why cosine-weighted sampling is so effective for diffuse materials—it eliminates one source of variance in the estimator.

1.5.3 Russian Roulette

Russian roulette provides an elegant solution to the path termination problem in Monte Carlo ray tracing, ensuring unbiased results while preventing infinite recursion. Named after the infamous game of chance, this technique randomly terminates paths based on their expected contribution, maintaining mathematical correctness while dramatically improving efficiency. Without Russian roulette, we face an uncomfortable dilemma: either truncate paths at a fixed depth (introducing bias) or trace paths indefinitely (impossible in practice). Russian roulette offers a third way—probabilistic termination that preserves the expected value while bounding computation.

The fundamental principle of Russian roulette is surprisingly simple: we randomly decide whether to continue tracing a path, and if we do continue, we increase the path's weight to compensate for the terminated paths. Specifically, if we continue a path with probability p, we must multiply its contribution by $1/p$ to maintain the correct expected value:

$$E[L_{RR}] = p \cdot \frac{L}{p} + (1-p) \cdot 0 = L \tag{46}$$

This shows that Russian roulette produces an unbiased estimate—the expected value equals the true value L regardless of the continuation probability p.

The continuation probability can be chosen in various ways, each with different trade-offs between bias, variance, and efficiency:

Fixed Probability Russian Roulette: The simplest approach uses a constant continuation probability after a certain depth:

```
if (depth > MIN_DEPTH) {
    float p_continue = 0.9;
    if (rng_float(rng) > p_continue) {
        return vec3_create(0, 0, 0);  // Terminate
    }
    throughput /= p_continue;  // Increase weight
}
```

While straightforward, this approach doesn't adapt to the path's actual contribution, potentially terminating important paths while continuing negligible ones.

Throughput-Based Russian Roulette: A more sophisticated approach bases the continuation probability on the path's throughput—the accumulated product of BRDFs and cosine terms:

$$p = \min(1, \max(\text{throughput.r}, \text{throughput.g}, \text{throughput.b})) \tag{47}$$

This adaptive strategy naturally terminates paths that have been heavily attenuated while allowing important paths to continue. Paths carrying significant energy are never terminated ($p = 1$ when throughput is high), while dim paths are terminated with probability proportional to their contribution.

The relationship between Russian roulette and variance requires careful consideration. While Russian roulette maintains the correct expected value, it does increase variance—some paths are terminated early (contributing zero), while others are boosted (contributing more than their natural value). The variance increase is:

$$\text{Var}[L_{RR}] = \frac{1-p}{p} L^2 + \text{Var}[L] \tag{48}$$

This additional variance term $(1-p)/p \cdot L^2$ shows why we should avoid terminating high-contribution paths—doing so would add substantial variance to our estimate.

Russian roulette becomes particularly important in scenes with high albedo materials or many reflective surfaces, where paths can bounce many times before naturally terminating. Consider a white room (albedo near 1.0) illuminated by a small light. Without Russian roulette, paths could bounce dozens or hundreds of times, each bounce contributing a small amount to the final image. Russian roulette allows us to statistically sample these long paths without explicitly tracing every bounce, dramatically reducing computation while maintaining correctness.

The implementation of Russian roulette must carefully handle edge cases and numerical precision:

```
vec3 trace_ray(Ray ray, Scene* scene, RNG* rng, int depth, vec3 throughput) {
    // Always trace a minimum number of bounces without RR
    const int RR_START_DEPTH = 3;

    if (depth >= MAX_DEPTH) {
        return vec3_create(0, 0, 0);  // Hard limit
    }

    HitRecord hit;
    if (!scene_intersect(scene, &ray, &hit)) {
        return scene->background;
    }

    // Handle emission
    vec3 emission = material_emission(&hit);

    // Russian roulette after minimum depth
    float p_continue = 1.0f;
    if (depth >= RR_START_DEPTH) {
        p_continue = fminf(1.0f, vec3_max_component(throughput));
        if (rng_float(rng) > p_continue) {
            return emission;  // Terminate, return only emission
        }
    }

    // Sample new direction
    vec3 brdf;
    Ray scattered;
    if (!material_scatter(&hit, &ray, &brdf, &scattered, rng)) {
        return emission;  // No scatter, return emission only
    }

    // Recursive trace with adjusted throughput
    vec3 new_throughput = vec3_div(vec3_mul(throughput, brdf), p_continue);
    vec3 incoming = trace_ray(scattered, scene, rng, depth + 1, new_throughput);

    return vec3_add(emission, vec3_mul(brdf, incoming) / p_continue);
}
```

1.5.4 Variance Reduction Techniques

Variance reduction techniques form the arsenal of methods that transform Monte Carlo rendering from a theoretical curiosity into a practical tool for producing high-quality images. While the basic Monte Carlo method guarantees convergence, the slow $1/\sqrt{N}$ rate means that naive implementations require enormous numbers of samples to achieve acceptable quality. Variance reduction techniques attack this problem from multiple angles, each exploiting different aspects of the rendering problem to reduce noise without introducing bias. Understanding and implementing these techniques can improve rendering efficiency by orders of magnitude.

Stratified Sampling divides the sampling domain into non-overlapping regions (strata) and ensures that each stratum receives a proportional number of samples. Instead of purely random sampling, which can leave gaps and create clusters, stratified sampling provides better coverage of the integration domain. For pixel sampling, we divide each pixel into a grid of subpixels and place one sample in each:

```
vec3 sample_pixel_stratified(int x, int y, int sx, int sy,
                             int strata_x, int strata_y, RNG* rng) {
    float u = (x + (sx + rng_float(rng)) / strata_x) / image_width;
    float v = (y + (sy + rng_float(rng)) / strata_y) / image_height;
    return camera_get_ray(u, v);
}
```

Stratified sampling reduces variance by eliminating the clustering that can occur with pure random sampling. The variance reduction factor is approximately $(\sigma^2_{strata}/\sigma^2_{random}) \approx 1/n$ for n strata when the integrand is smooth, though the benefit diminishes for discontinuous functions.

Low-Discrepancy Sequences (quasi-Monte Carlo) replace random numbers with carefully constructed deterministic sequences that provide better uniformity than random sampling. Sequences like Halton, Hammersley, and Sobol distribute points more evenly across the sampling domain:

$$\text{Halton}_b(i) = \sum_{j=0}^{\infty} d_j b^{-j-1} \tag{49}$$

where d_j are the digits of i in base b. These sequences achieve variance reduction through better spatial distribution, often providing $O((\log N)^d/N)$ convergence instead of $O(1/\sqrt{N})$ for smooth integrands.

Multiple Importance Sampling (MIS) combines multiple sampling strategies optimally when no single strategy works well everywhere. Consider direct lighting: we could sample the BRDF (good for specular surfaces) or sample the light source (good for small lights). MIS combines both strategies using carefully chosen weights that prevent high-variance samples from dominating:

$$L = \sum_{i=1}^{n_f} w_f(X_{f,i}) \frac{f(X_{f,i})}{p_f(X_{f,i})} + \sum_{j=1}^{n_g} w_g(X_{g,j}) \frac{f(X_{g,j})}{p_g(X_{g,j})} \tag{50}$$

The balance heuristic provides near-optimal weights:

$$w_f(x) = \frac{n_f p_f(x)}{n_f p_f(x) + n_g p_g(x)} \tag{51}$$

Control Variates reduce variance by subtracting a correlated function whose integral we know analytically. If we can find a function $g(x)$ similar to $f(x)$ with known integral I_g, we can estimate:

$$I_f = I_g + \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i) - g(x_i)}{p(x_i)} \tag{52}$$

The variance of $(f - g)$ is typically much smaller than the variance of f alone, leading to faster convergence.

Adaptive Sampling allocates computational effort based on local image statistics, placing more samples in regions with high variance. A simple implementation tracks the variance of pixel estimates and continues sampling until the variance falls below a threshold:

```
typedef struct {
    vec3 sum;
    vec3 sum_squares;
    int count;
} PixelStats;

float pixel_variance(PixelStats* stats) {
    vec3 mean = vec3_div(stats->sum, stats->count);
    vec3 mean_squares = vec3_div(stats->sum_squares, stats->count);
    vec3 variance = vec3_sub(mean_squares, vec3_mul(mean, mean));
    return vec3_max_component(variance);
}

void adaptive_sample_pixel(int x, int y, float threshold) {
    PixelStats stats = {0};

    while (stats.count < MIN_SAMPLES ||
          (stats.count < MAX_SAMPLES &&
           pixel_variance(&stats) > threshold)) {
        vec3 sample = trace_pixel(x, y);
        stats.sum = vec3_add(stats.sum, sample);
        stats.sum_squares = vec3_add(stats.sum_squares,
                                     vec3_mul(sample, sample));
        stats.count++;
    }

    set_pixel(x, y, vec3_div(stats.sum, stats.count));
}
```

Importance Resampling generates better distributed samples by resampling from an initial set based on their importance. This technique is particularly useful when the optimal sampling distribution is unknown but can be estimated from samples:

1. Generate M preliminary samples $x_1, ..., x_M$ uniformly
2. Compute importance weights $w_i = f(x_i)/p(x_i)$
3. Resample N samples from the preliminary set with probability proportional to w_i
4. Use the resampled set for the final estimate

Bidirectional Techniques trace paths from both the camera and light sources, connecting them to form complete light paths. While more complex to implement than unidirectional path tracing, bidirectional methods excel at capturing certain lighting effects:

- Caustics (light focused through glass) are easily captured by light paths but difficult for eye paths
- Indirect illumination of diffuse surfaces is efficiently sampled by eye paths
- Direct illumination can be computed by both strategies and combined via MIS

The effectiveness of variance reduction techniques depends heavily on the specific rendering scenario. Glossy reflections benefit from good BRDF sampling, small light sources require explicit light sampling, and high-frequency textures need careful filtering. The art of efficient rendering lies in choosing and combining appropriate variance reduction techniques for the scene at hand, balancing implementation complexity against performance gains. Modern production renderers employ dozens of variance reduction techniques, carefully tuned and combined to achieve the performance necessary for feature film production.

1.6 BVH Acceleration Structure

The computational bottleneck in any ray tracer lies in the ray-primitive intersection tests. A naive implementation that tests every ray against every primitive in the scene exhibits O(N) complexity per ray, where N is the number of primitives. For a scene with millions of triangles and billions of rays, this quadratic scaling becomes prohibitively expensive. Acceleration structures transform this linear search into a logarithmic one, reducing intersection tests from millions to mere dozens per ray. Among the various acceleration structures developed over decades of research—uniform grids, octrees, kd-trees, and bounding volume hierarchies—the BVH has emerged as the de facto standard for production ray tracing due to its robust performance across diverse scenes, straightforward construction, and efficient memory usage.

1.6.1 Spatial Acceleration Structures

Spatial acceleration structures organize scene geometry to enable rapid rejection of large groups of primitives that cannot possibly intersect with a given ray. The fundamental principle underlying all acceleration structures is the exploitation of spatial coherence: primitives near each other in 3D space are grouped together, allowing entire groups to be culled with a single test. This hierarchical culling transforms the ray tracing problem from testing every primitive to traversing a tree structure, where each node test eliminates entire subtrees from consideration.

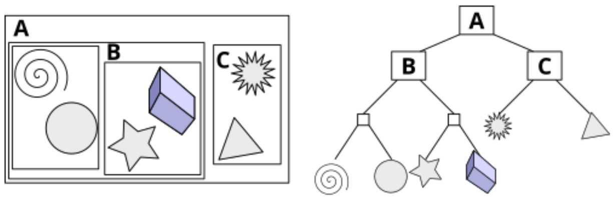
The landscape of acceleration structures can be broadly categorized into spatial subdivisions and object hierarchies. Spatial subdivision schemes like uniform grids, octrees, and kd-trees partition space into regions, with primitives assigned to the regions they overlap. These structures excel at handling uniformly distributed geometry but struggle with varying primitive densities—a common characteristic of real-world scenes where detail varies dramatically across the model. Object hierarchies like BVHs, by contrast, adapt to the geometry distribution by grouping primitives based on their spatial proximity rather than fixed space partitions. This adaptive nature makes BVHs particularly robust for production use, handling everything from architectural models with vast empty spaces to dense vegetation with millions of leaves.

The choice of acceleration structure profoundly impacts both build time and traversal performance. Grid structures offer O(1) build time but suffer from poor traversal performance in scenes with non-uniform primitive distributions. Kd-trees provide excellent traversal performance through adaptive spatial splitting but require complex construction algorithms and can suffer from numerical robustness issues at primitive boundaries. BVHs strike a practical balance: reasonable build times, robust traversal performance, and straightforward implementation. Moreover, BVHs naturally support dynamic scenes through refitting operations, making them suitable for animated content where rebuilding the entire structure every frame would be prohibitive.

Memory considerations further favor BVHs in production environments. While grids can require excessive memory for sparse scenes and kd-trees need careful memory management for split primitives, BVHs maintain a simple one-to-one relationship between primitives and leaf nodes. Each primitive appears exactly once in the hierarchy, simplifying memory management and enabling straightforward parallel construction algorithms. The tree structure itself requires only two child pointers and a bounding box per internal node, resulting in predictable memory usage of approximately 2N-1 nodes for N primitives.

1.6.2 Bounding Volume Hierarchy (BVH)

A Bounding Volume Hierarchy organizes geometric primitives into a tree structure where each node contains a bounding volume that encloses all geometry in its subtree. The elegance of the BVH lies in its conceptual simplicity: if a ray doesn't intersect a node's bounding volume, it cannot intersect any geometry within that node's subtree. This observation enables rapid culling of large portions of the scene with simple ray-box intersection tests, which are significantly faster than ray-triangle or ray-sphere tests. The hierarchical nature means that successful culling at higher tree levels eliminates exponentially more primitives from consideration.



The choice of bounding volume shape represents a fundamental design decision in BVH construction. Axis-aligned bounding boxes (AABBs) have become the universal choice due to their optimal balance of tightness and computational efficiency. Ray-AABB intersection requires only six comparisons and can be implemented without branches on modern processors. Alternative bounding volumes like oriented bounding boxes (OBBs) or spheres might provide tighter bounds for certain geometry configurations but require more expensive intersection tests that negate any traversal savings. The axis-aligned constraint of AABBs, while sometimes producing looser bounds for diagonal or randomly oriented geometry, is more than compensated by the efficiency of intersection testing.

The tree topology of a BVH directly impacts traversal performance. Binary BVHs, where each internal node has exactly two children, have become standard due to their simplicity and good cache behavior. Some implementations explore wider branching factors (4-way or 8-way BVHs) to reduce tree depth, but these often suffer from increased node intersection costs and poorer cache utilization. The binary structure also maps naturally to parallel construction algorithms and SIMD traversal implementations, where pairs of child nodes can be tested simultaneously using vector instructions.

Quality metrics for BVH construction focus on minimizing the expected cost of ray traversal. The Surface Area Heuristic (SAH), which models the probability of ray-box intersection as proportional to surface area, has proven remarkably effective at predicting traversal cost. A well-constructed BVH using SAH typically reduces intersection tests by orders of magnitude compared to naive testing, with deeper trees generally providing better culling at the cost of increased traversal overhead. The sweet spot typically occurs when leaf nodes contain 1-4 primitives, balancing traversal depth against the cost of primitive intersection tests.

1.6.3 BVH Construction

The construction of an efficient BVH represents a classic optimization problem: how to partition primitives into a hierarchy that minimizes the expected cost of ray traversal. While optimal BVH construction is NP-hard, practical heuristics produce near-optimal trees in reasonable time. The Surface Area Heuristic has emerged as the gold standard, providing a principled cost model that correlates strongly with actual traversal performance.

Surface Area Heuristic (SAH):

The SAH cost model estimates traversal cost as:

$$C = C_{traverse} + P_{left} \cdot C_{left} + P_{right} \cdot C_{right} \tag{53}$$

where $C_{traverse}$ represents the cost of visiting the current node, and P_{left} and P_{right} represent the probabilities of a ray intersecting the left and right child bounding boxes, respectively. These probabilities are approximated using the surface areas of the bounding boxes, based on the observation that uniformly distributed random rays intersect boxes with probability proportional to their surface area. This geometric probability model, while making assumptions about ray distributions that don't always hold in practice, provides robust performance across a wide variety of scenes and viewpoints.

The implementation of SAH-based construction typically follows a top-down recursive approach. Starting with all primitives in the root node, the algorithm considers multiple candidate splitting planes along each axis. For each candidate split, it computes the SAH cost of the resulting partition. The split with minimum cost is selected, and the process recurses on the two child nodes. The choice of candidate splits significantly impacts both build quality and construction time. Common strategies include testing splits at primitive centroids, uniform spatial divisions, or the full set of primitive bounding box extents. The latter, while computationally expensive, often produces the highest quality trees.

Practical implementations must balance tree quality against construction time. Full SAH evaluation with all possible splits produces excellent trees but can be prohibitively expensive for complex scenes. Approximate strategies like binned SAH reduce the candidate splits to a fixed number of spatial bins (typically 16-32), providing most of the quality benefit at a fraction of the cost. For extremely large scenes or real-time applications, even faster methods like median splitting or linear BVH construction may be employed, trading tree quality for construction speed. The choice of construction algorithm depends on whether the BVH is built once for multiple frames (favoring quality) or rebuilt every frame for dynamic scenes (favoring speed).

2 Code Flow and Implementation Tasks

2.1 Code Flow Overview

The provided C codebase implements a complete path tracing framework with GTK3 GUI. The architecture cleanly separates implemented infrastructure (GUI, threading, memory management) from student tasks (ray tracing algorithms, materials, intersections, BVH), allowing students to focus on graphics algorithms without system programming complexities.

2.1.1 Architecture and Module Organization

The codebase consists of three layers:

- **GUI Layer** (`gui.c`): Handles user interaction and scene selection
- **Rendering Layer**: Manages OpenMP parallelization across CPU cores
- **Path Tracing Engine**: Core algorithms in `pathtracer.c`, `material.c`, `primitive.c`, and `bvh.c`

When rendering begins, the GUI spawns a background thread to prevent UI freezing. This thread uses OpenMP to parallelize pixel computation, with each thread independently tracing rays. Multiple samples per pixel provide antialiasing through the `trace_ray` function, which recursively follows light paths. Results are averaged and saved as BMP files.

Memory is managed through centralized allocation in scene creation and BVH construction. Stack allocation for rays and vectors avoids manual memory management in performance-critical code. Thread-local random number generators prevent contention.

2.1.2 Key Data Structures and Their Relationships

Core data structures:

- `Scene`: Container for primitives, materials, and camera configuration
- `Primitive`: Geometric data (sphere: center/radius, triangle: three vertices) with material index
- `Material`: Surface properties (type, albedo, roughness/IOR)
- `Ray`: Origin and direction vectors for light paths
- `HitRecord`: Intersection data (point, normal, material, parameter t)

- `BVHNode`: Tree node with AABB and child pointers or primitive index

The `vec3` type handles 3D math operations (dot, cross, normalize) with SIMD optimization handled transparently by the framework.

2.1.3 Control Flow During Rendering

Rendering flow:

1. User clicks "Render" → GUI disables controls and spawns background thread
2. Background thread initiates OpenMP parallel region
3. Each CPU core processes pixels independently with dynamic scheduling
4. Per pixel: Generate multiple samples with random offsets for antialiasing
5. Each sample calls `trace_ray` (student implementation) which:
    - Finds ray-scene intersection using BVH
    - Evaluates material properties at hit points
    - Computes scattered rays based on BRDF
    - Recursively traces with Russian roulette termination
    - Accumulates emission from light sources
6. Average samples and save as BMP file

2.2 Implementation Tasks

This assignment consists of four main implementation tasks that build upon each other to create a complete path tracer. Each task has been carefully designed to teach specific concepts in physically-based rendering while maintaining reasonable implementation complexity. Students should implement these tasks in order, as later tasks depend on earlier ones. The provided framework handles all system-level complexities, allowing you to focus entirely on the graphics algorithms.

You will be working in four main files:

- `primitive.c`: Ray-primitive intersection algorithms (Task 1)
- `material.c`: Material scattering functions (Task 2)
- `pathtracer.c`: Main path tracing algorithm (Task 3)
- `bvh.c`: BVH construction and traversal (Task 4)

Total points for this assignment: 100 points, distributed across the core path tracing components.

2.2.1 Task 1: Ray-Primitive Intersection (25 points) - File: primitive.c

The foundation of any ray tracer is the ability to determine where rays intersect with geometric primitives. In this task, you will implement intersection algorithms for spheres and triangles in `primitive.c`. These algorithms must be both correct and efficient, as they will be called billions of times during a typical render. The mathematical concepts were covered in the theory section, and now you'll translate those equations into working code.

Subtask 1.1: Ray-Sphere Intersection (12 points)

Implement the `sphere_hit` function in `primitive.c`. This function takes a ray, a sphere, and a valid t range, returning whether an intersection exists and filling in the hit record if it does. Your implementation should:

- Solve the quadratic equation derived from substituting the ray equation into the sphere equation
- Handle the discriminant correctly: negative means no intersection, zero means one tangent intersection, positive means two intersections
- Choose the correct t value: the smallest positive t within the valid range [t_min, t_max]
- Compute the hit point using the ray equation: $\mathbf{p} = \mathbf{o} + t\mathbf{d}$
- Calculate the outward-facing normal: $\mathbf{n} = (\mathbf{p} - \mathbf{c})/r$ where c is center and r is radius
- Determine if the ray hit the front or back face by checking if the ray direction and normal point in opposite directions
- Set all fields in the hit record structure correctly

Subtask 1.2: Ray-Triangle Intersection (13 points)

Implement the `triangle_hit` function using the Möller-Trumbore algorithm. This efficient algorithm simultaneously computes the intersection point and barycentric coordinates without requiring pre-computation of the plane equation. Your implementation should:

- Compute edge vectors: $\mathbf{e_1} = \mathbf{v_1} - \mathbf{v_0}$ and $\mathbf{e_2} = \mathbf{v_2} - \mathbf{v_0}$
- Calculate the determinant to check if the ray is parallel to the triangle plane
- Use a small epsilon (e.g., 1e-8) to handle near-parallel cases
- Compute barycentric coordinates u and v, checking that $u \geq 0$, $v \geq 0$, and $u + v \leq 1$
- Calculate the t parameter and verify it's within the valid range
- Compute the intersection point and normal using the cross product of edge vectors
- Handle both front and back face intersections correctly

The algorithm should early-exit when any constraint is violated to maximize efficiency. Pay attention to the order of operations to minimize computational cost for rays that don't intersect the triangle.

2.2.2 Task 2: Material Scattering (25 points) - File: material.c

Materials determine how light interacts with surfaces, encoding the visual appearance of objects. In this task, you'll implement three fundamental material types in `material.c`. Each material must correctly sample the BRDF and compute the probability density function (PDF) for importance sampling. The scattering functions must be energy-conserving to ensure physical correctness.

Subtask 2.1: Lambertian Diffuse Material (8 points)

Implement Lambertian scattering in the `MATERIAL_LAMBERTIAN` case of `material_scatter`. Lambertian surfaces exhibit perfect diffuse reflection, scattering light equally in all directions above the surface. Your implementation should:

- Generate a random direction in the hemisphere above the surface using cosine-weighted sampling
- Use the provided `random_unit_vector()` and ensure the scattered direction is in the same hemisphere as the normal

- Handle the degenerate case where the random vector is exactly opposite to the normal
- Set the scattered ray origin at the hit point (offset slightly along the normal to avoid self-intersection)
- Set the attenuation to the material's albedo color
- Return true to indicate successful scattering

The cosine-weighted sampling is crucial for variance reduction. A common approach is to generate a random unit vector and add it to the normal, then normalize the result. This naturally produces a cosine-weighted distribution.

Subtask 2.2: Metal Material with Roughness (8 points)

Implement metal scattering with controllable roughness. Metals exhibit specular reflection that can be perfect (mirror-like) or rough depending on surface properties. Your implementation should:

- Compute the mirror reflection direction: $\mathbf{r} = \mathbf{v} - 2(\mathbf{v} \cdot \mathbf{n})\mathbf{n}$ where v is the incident direction
- Add roughness by perturbing the reflected direction with a random vector scaled by the roughness parameter
- Ensure the scattered ray is in the correct hemisphere (dot product of scattered direction and normal should be positive)
- Set the attenuation to the material's albedo (representing the metal's color)
- Return false if the scattered ray goes below the surface (absorbed)

The roughness parameter should range from 0 (perfect mirror) to 1 (very rough). Be careful with the sign conventions for incident and reflected directions.

Subtask 2.3: Dielectric Material with Refraction (9 points)

Implement dielectric (transparent) materials that exhibit both reflection and refraction according to Fresnel equations. This is the most complex material type, requiring careful handling of total internal reflection and Fresnel coefficients. Your implementation should:

- Determine if the ray is entering or exiting the material by checking the face orientation
- Calculate the refraction ratio: either η_1/η_2 (entering) or η_2/η_1 (exiting)
- Compute the cosine of the incident angle
- Check for total internal reflection: when $\sin\theta_t > 1$ where $\sin\theta_t = \frac{\eta_i}{\eta_t}\sin\theta_i$
- Use Schlick's approximation for the Fresnel reflectance coefficient
- Randomly choose between reflection and refraction based on the Fresnel coefficient
- Compute the refracted direction using Snell's law when refraction occurs
- Set attenuation to white (vec3(1,1,1)) as dielectrics don't absorb light in our simple model

This material type requires the most careful implementation due to the many edge cases and the critical nature of total internal reflection for effects like caustics.

2.2.3 Task 3: Path Tracing Integration (25 points) - File: pathtracer.c

The heart of the path tracer is the recursive ray tracing function that follows light paths through the scene. This task requires integrating all previous components into a cohesive Monte Carlo path tracer in `pathtracer.c`. You'll implement the main rendering equation solver that handles recursive bouncing, Russian roulette termination, and light accumulation. This is where the theory of path tracing becomes practical code.

Implementation Requirements:

Implement the `trace_ray` function in `pathtracer.c` with the following logic:

- Check recursion depth and implement Russian roulette for unbiased termination:
  - For depth ¿ RUSSIAN_ROULETTE_DEPTH, compute survival probability based on ray throughput
  - Use the maximum color component as survival probability (clamped to a reasonable maximum like 0.95)
  - Randomly terminate paths based on this probability
  - Scale the contribution of surviving paths by $1/p$ to maintain the correct expected value
- Find the closest intersection with the scene:
  - Use the BVH traversal function (which you'll implement in Task 4)
  - If no intersection, return the background color (could be sky gradient or constant)
  - Track the closest hit distance to early-terminate BVH traversal
- Handle emissive materials:
  - Check if the hit material is emissive
  - Add emission contribution to the accumulated light
  - Emissive materials should not scatter rays (they only emit, not reflect)
- Generate scattered rays:
  - Call the material scattering function
  - If scattering fails (absorbed), return the accumulated emission
  - Otherwise, recursively trace the scattered ray
- Accumulate contributions:
  - Multiply the recursive result by the material attenuation
  - Add any emission from the current surface
  - Apply the cosine term implicitly handled by importance sampling

The function should handle edge cases gracefully: rays that immediately leave the scene, numerical precision issues at surface intersections, and degenerate scattering directions. Use a small epsilon offset when spawning secondary rays to avoid self-intersection artifacts.

2.2.4 Task 4: BVH Acceleration Structure (25 points) - File: bvh.c

The Bounding Volume Hierarchy dramatically accelerates ray tracing by organizing primitives into a tree structure that allows rapid culling of non-intersecting geometry. Without BVH, rendering complex scenes would be impractically slow. This task involves both building the tree structure and efficiently traversing it during ray tracing in `bvh.c`. The Surface Area Heuristic (SAH) guides the construction to minimize expected traversal cost.

Subtask 4.1: BVH Construction with SAH (13 points)

Implement the `bvh_build_recursive` function that constructs the BVH tree using the Surface Area Heuristic. Your implementation should:

- Base case: Create a leaf node when the primitive count is below a threshold (typically 1-4 primitives)
- Compute the bounding box that encloses all primitives in the current node
- Try splitting along each axis at multiple candidate positions:
  - Use binned SAH with 12-16 bins for efficiency
  - For each bin boundary, compute the cost of splitting there
  - SAH cost $= C_{traverse} + \frac{A_{left}}{A_{node}} \cdot N_{left} \cdot C_{intersect} + \frac{A_{right}}{A_{node}} \cdot N_{right} \cdot C_{intersect}$
  - Where A is surface area and N is primitive count
  - Use typical constants: $C_{traverse} = 1.0$, $C_{intersect} = 1.0$
- Select the split with minimum cost
- Partition primitives based on their centroid position relative to the split
- Recursively build left and right subtrees
- Handle edge cases: all primitives on one side, identical primitive bounds

The quality of the BVH directly impacts rendering performance, so careful implementation of SAH is crucial. Consider falling back to median splitting when SAH doesn't find a good partition.

Subtask 4.2: BVH Traversal (12 points)

Implement the `bvh_hit` function that efficiently traverses the BVH to find ray-primitive intersections. Your traversal should:

- Start at the root node with a traversal stack
- For each node:
  - Test ray-AABB intersection using the provided `aabb_hit` function
  - If no intersection, skip this subtree entirely
  - For leaf nodes: test the ray against the contained primitive
  - For internal nodes: add children to the traversal stack
- Implement ordered traversal:
  - Determine which child is closer along the ray
  - Visit closer child first for better early termination
  - Use ray direction signs to avoid computing distances

- Maintain closest hit distance:
  - Update t_max whenever a closer hit is found
  - Use this to cull nodes that are farther than the current closest hit
- Use an explicit stack rather than recursion:
  - Avoids function call overhead
  - Prevents stack overflow for deep trees
  - Enables better control over traversal order

Efficient traversal is critical for performance. Even small optimizations in this function can have significant impact on overall rendering time. Pay attention to memory access patterns and minimize redundant calculations.

2.2.5 Testing and Validation

To test your implementation, compile and run the path tracer:

```
make release          # Compile with optimizations
./pathtracer_gui      # Launch the GUI application
```

Once the GUI opens, select different scenes from the dropdown menu and click "Render" to test your implementation using provided scenes.

Compare your results with the reference implementation shown in this video:
`https://youtu.be/AVhjcjf5UHM`.

For Windows Users:
Windows users should use Windows Subsystem for Linux (WSL) to run this project:

- Install WSL2 with Ubuntu distribution
- Install required dependencies as described in the template's `README.md`

Common Issues:

- If rendering is extremely slow, verify your BVH implementation
- If images are black, check your emission handling and path tracing recursion
- If materials look wrong, verify normal calculations and material scattering
- Noise in images is expected - increase samples per pixel for cleaner results

3 Submission Format

Submit your source code as a single ZIP file on SCeLE and be prepared for a **live demo** where your team explains and demonstrates your implementation. This assignment is done in **pairs**.

ZIP Archive

- **Filename:** `<NPM1>-<NPM2>-A3-raytracer.zip`

Live Demo (pair)

- **Schedule:** TBA by the TAs on SCeLE.

Happy Rendering!