RayTracer: Rendering of Complex Scenes

An Advanced Guidance of RayTracer based on
Ray Tracing in One Weekend Series

WyEhNf

July 2026

Contents

Chapter 0 Introduction

This report was written during the 2026 ACM Class PPCA period. Based on the renowned Ray Tracing in One Weekend series, the author implemented a RayTracer in Rust. This report documents the knowledge acquired and practices undertaken during the project. It also serves as a reference for the refinement and further development of subsequent projects. Before reading this report, readers are advised to first study the aforementioned classic series to acquire foundational knowledge of computer graphics. The main content of this report focuses on GPU-side ray tracing implementation architecture and rendering techniques for relatively complex natural materials, offered as a possible extension and elaboration.

The implementation of this project was completed with the assistance of a Code Agent. In the rendering of complex scenes, the required code can be relatively long and complex; making good use of a Code Agent to support code implementation helps improve development efficiency and continuity.

Before the article begins, the author wishes to express gratitude and respect to the teaching assistants who devoted their efforts to this project.

1 The Rendering Equation

1.1 Radiance and Light Transport

Light propagation in a scene can be described by radiance. The radiance $L(\mathbf{p}, \omega)$ represents the radiant flux density at a spatial point $\mathbf{p}$ in direction ω, measured in $\mathrm{W}/(\mathrm{m}^2 \cdot \mathrm{sr})$. In a vacuum, radiance remains constant along a ray when absorption and scattering by participating media are absent.

When light encounters a surface, the incident radiance is scattered by the surface, with a portion leaving in the outgoing direction. This scattering process constitutes the core of the rendering equation. The outgoing radiance $L_o(\mathbf{p}, \omega_o)$ from surface point $\mathbf{p}$ in direction ω_o consists of two components: the radiance emitted by the point itself $L_e(\mathbf{p}, \omega_o)$, and the contribution of incident radiance scattered by the surface toward the outgoing direction. The latter is an integral over the upper hemisphere H^2: for all incident directions ω_i, the incident radiance $L_i(\mathbf{p}, \omega_i)$ is weighted by the bidirectional reflectance distribution function (BRDF) $f_r(\mathbf{p}, \omega_i, \omega_o)$ and multiplied by the cosine of the angle between the incident direction and the surface normal, $\cos\theta_i = \mathbf{n} \cdot \omega_i$. This cosine factor arises from the definition of irradiance: the radiant flux received per unit area is proportional to the cosine of the incident angle; grazing light spreads over a larger surface area, so its contribution per unit area is smaller. Hence:

$$L_o(\mathbf{p}, \omega_o) = L_e(\mathbf{p}, \omega_o) + \int_{H^2} f_r(\mathbf{p}, \omega_i, \omega_o) \cdot L_i(\mathbf{p}, \omega_i) \cdot \cos\theta_i \, d\omega_i \tag{1}$$

This equation was proposed by Kajiya in 1986 and unifies all light transport processes in a scene into a single integral equation. The BRDF f_r encodes the material properties of the surface—it determines the distribution of incident light across outgoing directions. For an ideal diffuse surface (Lambertian), f_r is constant in all outgoing directions; for a mirror, f_r is nonzero only near the reflection direction; for dielectrics, f_r must also account for the energy distribution in the refracted direction.

The $L_i(\mathbf{p}, \omega_i)$ term itself depends on the outgoing radiance of other surfaces in the scene toward $\mathbf{p}$—tracing backward along $-\omega_i$, the outgoing radiance at the first intersected surface point $\mathbf{p}'$ is precisely the incident radiance at $\mathbf{p}$ from direction ω_i. This recursive relationship makes the rendering equation a Fredholm integral equation of the second kind, whose solution describes the steady-state radiance distribution across all surfaces in all directions in the scene.

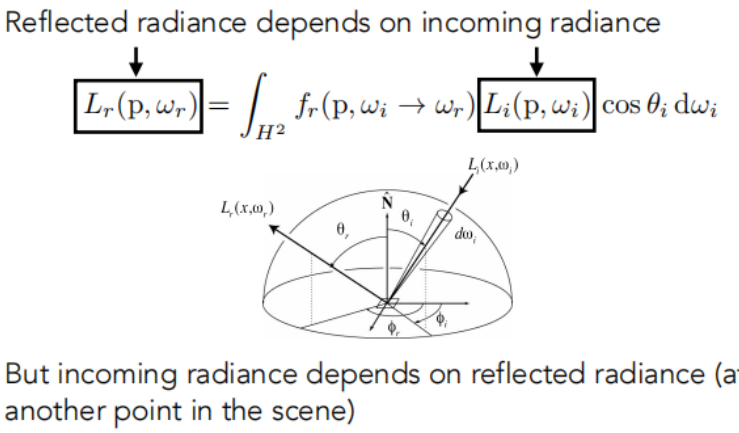

Figure 1: Geometric illustration of the rendering equation: the incident radiance L_i at surface point $\mathbf{p}$ comes from the outgoing radiance of other surfaces in the scene along direction $-\omega_i$, modulated by the BRDF f_r to contribute toward ω_o.

1.2 Monte Carlo Integration

The hemispherical integral in the rendering equation generally has no analytic solution—the integrand involves arbitrary BRDFs and recursively defined incident radiance, whose complexity renders closed-form integration infeasible. Monte Carlo methods provide a numerical approximation.

Let the integral to be computed be $I = \int_\Omega f(x)\,dx$. Choose a probability density function (PDF) $p(x)$ defined on Ω, satisfying $p(x) > 0$ for all points where $f(x) \neq 0$. Draw N independent samples $X_1, \ldots, X_N$ from $p(x)$ and construct the estimator:

$$\langle I^N \rangle = \frac{1}{N} \sum_{k=1}^{N} \frac{f(X_k)}{p(X_k)} \tag{2}$$

This estimator is unbiased. Taking the expectation:

$$\mathbb{E}\left[\frac{f(X)}{p(X)}\right] = \int_\Omega \frac{f(x)}{p(x)} \cdot p(x)\,dx = \int_\Omega f(x)\,dx = I \tag{3}$$

The operation of dividing by $p(x)$ in the integrand exactly cancels the sampling bias introduced by drawing samples from $p(x)$—regions with high density $p(x)$ are sampled more frequently, and dividing by large $p(x)$ reduces the weight of those samples; conversely, samples from sparse regions receive amplified weight. This is the essential distinction between Monte Carlo integration and simple random sampling.

The variance of the estimator is:

$$\mathrm{Var}[\langle I^N \rangle] = \frac{1}{N} \mathrm{Var}\left[\frac{f(X)}{p(X)}\right] \tag{4}$$

The variance formula reveals three critical facts. First, the variance of the estimator is proportional to the single-sample variance and inversely proportional to N—the root mean square error decays at a rate of $1/\sqrt{N}$. Second, the variance is independent of the dimension of the integration domain Ω—the integrand f and the sampling density p map to a scalar ratio in any dimension, and the statistical behavior of the estimator depends only on the distribution of this scalar random variable. Third, the variance can be reduced by choosing $p(x)$ whose shape approaches that of $f(x)$—the closer the ratio f/p is to a constant, the smaller the single-sample variance; if $p(x) \propto f(x)$, the variance becomes zero, and one sample yields the exact solution.

Dimension independence is the fundamental advantage of Monte Carlo methods over deterministic quadrature rules.

Consider a typical deterministic method: placing m sample points along each coordinate axis in a d-dimensional space, forming a d-dimensional tensor-product grid with a total of m^d sample points. In low dimensions ($d = 2$ or 3), such a grid is efficient and accurate. However, the dimension of the integration domain in the rendering equation grows with the light path length: a path with k bounces corresponds to a $2k$-dimensional integral—each scattering event introduces two directional variables (azimuth and polar angles). In such a high-dimensional space, the number of tensor-product grid points grows exponentially with dimension; at $d = 10$, even with only 10 points per dimension, 10^{10} samples are required, which is computationally infeasible. The error of Monte Carlo methods depends only on the sample count N rather than the dimension, giving them an overwhelming advantage for the high-dimensional integrals of the rendering equation.

Applying the Monte Carlo estimator to the rendering equation (1) yields the fundamental formula of path tracing:

$$\langle L_o \rangle = L_e + \frac{1}{N} \sum_{k=1}^{N} \frac{f_r(\omega_k) \cdot L_i(\omega_k) \cdot \cos\theta_k}{p(\omega_k)} \tag{5}$$

Each sample corresponds to shooting a ray from surface point $\mathbf{p}$ in a random direction ω_k—the radiance at the ray's intersection point is $L_i(\omega_k)$, which itself must be recursively computed via the same Monte Carlo estimation. This recursive chain forms a light path, and the average of N light paths is the Monte Carlo estimate of the pixel color.

1.3 Importance Sampling

The variance formula from the previous section indicates that when $p(x)$ approximates the shape of $f(x)$, the ratio f/p tends toward a constant, and the variance decreases accordingly. Conversely, if $p(x)$ bears no relation to the integrand, the variance can

be arbitrarily large. This fact has direct consequences in rendering.

Consider the most naive sampling strategy—uniformly selecting directions over the hemisphere, i.e., $p(\omega) = 1/(2\pi)$. The advantage of this choice is its simplicity and scene independence. But its defect is equally apparent: the vast majority of randomly generated directions point neither toward light sources nor toward the BRDF peak directions. In these directions, the contribution of $L_i(\omega_k)$ is negligible, yet they consume equal computational resources. Meanwhile, the few directions that happen to point toward light sources contribute enormously, but their contributions are diluted by the low probability of uniform sampling, causing the per-sample f/p values to fluctuate wildly—this is precisely the source of high variance. At low sample counts, this fluctuation manifests as noise in the image: some pixels happen to sample light-source directions and appear anomalously bright, while neighboring pixels miss the light source and appear dark.

The root cause of this problem is the severe mismatch between the uniformly distributed $p(x)$ and the shape of the integrand $f_r \cdot L_i \cdot \cos\theta$. The incident radiance L_i has sharp peaks in light-source directions; the BRDF f_r is concentrated near the specular reflection direction; the cosine factor $\cos\theta$ is maximal at the normal direction and approaches zero at grazing angles. The uniform PDF is entirely blind to these structures, wasting the vast majority of samples on directions with near-zero contribution.

Importance sampling is designed precisely to address this defect. Its idea is to make $p(\omega)$ approximate some easily sampleable factor of the integrand, so that f/p is nearly constant in any sampled direction. In practice, one typically chooses to approximate the product of the BRDF and the cosine factor $f_r \cdot \cos\theta$, since the shape of this part is determined by material parameters and does not change with the positions of light sources in the scene.

Taking Lambertian material as an example, $f_r = \text{albedo}/\pi$. Substituting into the integrand of Equation (5):

$$\frac{f_r \cdot L_i \cdot \cos\theta}{p} = \frac{(\text{albedo}/\pi) \cdot L_i \cdot \cos\theta}{p} \tag{6}$$

If the cosine-weighted PDF $p(\omega) = \cos\theta/\pi$ is chosen, the integrand simplifies to:

$$\frac{(\text{albedo}/\pi) \cdot L_i \cdot \cos\theta}{\cos\theta/\pi} = \text{albedo} \cdot L_i \tag{7}$$

The material color and cosine factor in the numerator are exactly canceled by the PDF in the denominator, leaving only the albedo multiplied by the recursively evaluated incident radiance. Compared to uniform sampling, cosine-weighted sampling concentrates more samples near the normal direction—precisely where a diffuse surface receives the most illumination—thereby significantly reducing the variance of the

estimator.

Having chosen a PDF, the next question is: how does one generate random direction samples from that PDF? For the cosine-weighted PDF $p(\theta, \phi) = \cos\theta/\pi$, in spherical coordinates $d\omega = \sin\theta\, d\theta\, d\phi$, so the marginal distribution of the probability density with respect to the polar angle θ is $p(\theta) = 2\cos\theta\sin\theta$ (after integrating over ϕ). The cumulative distribution function of this distribution is $P(\theta) = \sin^2\theta$. Inverting the CDF, the polar angle is generated from a uniform random number ξ: $\sin\theta = \sqrt{\xi}$, i.e., $\cos\theta = \sqrt{1-\xi}$, $r = \sin\theta = \sqrt{\xi}$. The azimuthal angle ϕ is uniformly distributed over $[0, 2\pi]$. This is known as Malley's method, and the directions it generates are distributed in a local coordinate system with the normal vector as the Z-axis.

After generating a direction in the local coordinate system, it must be transformed to the actual normal direction of the hit surface. Given a surface normal $\mathbf{n}$, an orthonormal basis (ONB) $\{\mathbf{u}, \mathbf{v}, \mathbf{w}\}$ is constructed with $\mathbf{w} = \mathbf{n}$, where $\mathbf{u}$ and $\mathbf{v}$ span the tangent plane. An auxiliary vector not parallel to $\mathbf{n}$ is chosen (taking $(0, 1, 0)$ when $\mathbf{n}$ is close to the X-axis, and $(1, 0, 0)$ otherwise), and the basis vectors are built via cross products: $\mathbf{u} = (\mathbf{a} \times \mathbf{n})/\|\mathbf{a} \times \mathbf{n}\|$, $\mathbf{v} = \mathbf{n} \times \mathbf{u}$. The local direction (x, y, z) is transformed to world space: $\omega_{\text{world}} = x\mathbf{u} + y\mathbf{v} + z\mathbf{n}$. This transformation is orthogonal (determinant 1), so a PDF defined in local space remains valid in world space without any Jacobian correction—$p_{\text{world}}(\omega_{\text{world}}) = p_{\text{local}}(\omega_{\text{local}})$.

This simplification applies not only to Lambertian surfaces but also reveals the general principle of importance sampling: for any BRDF, if a sampling density proportional to $f_r \cdot \cos\theta$ can be constructed, the variance introduced by the BRDF is eliminated, compressing the variance source to only the variation in L_i. For specular materials, this means sampling directions are concentrated within a narrow cone around the reflection direction; for glossy surfaces, the cone width increases with roughness.

The transition from uniform direction sampling to importance sampling is the critical step that makes path tracing go from "feasible" to "usable." Uniform sampling may be acceptable in scenes with simple lighting, but in scenes with small light sources or sharp BRDFs, the variance is so large that astronomical sample counts are required for convergence. Importance sampling automatically allocates sampling resources to the directions that contribute most to the integrand, making acceptable image quality achievable at tens to hundreds of samples per pixel (SPP).

1.4 Multiple Importance Sampling

Importance sampling solves the problem of matching a single PDF to the integrand, but rendering often faces another dilemma: the integrand has different "peak sources" in different regions of the integration domain. Taking direct illumination as an exam-

ple, the integrand $f_r \cdot L_i \cdot \cos\theta$ has a sharp peak in light-source directions (from the contribution of L_i) and simultaneously another peak near the BRDF reflection direction (from the contribution of $f_r \cdot \cos\theta$). If only a BRDF-oriented sampling strategy is used, most sample directions deviate from light sources, providing insufficient coverage of the L_i peak; if only a light-source-direction sampling strategy is used, performance is poor when the light source is occluded or when the BRDF is specular. Each strategy excels in its own region but fails in the other; no single strategy can cover both.

Multiple importance sampling (MIS), proposed by Veach in 1995, addresses this by combining multiple sampling strategies into a single mixture estimator. Given M sampling strategies, where the k-th strategy has PDF $p_k(\omega)$, the mixture PDF is defined as their convex combination:

$$p_{\text{mix}}(\omega) = \sum_{k=1}^{M} w_k \cdot p_k(\omega), \quad \sum_{k=1}^{M} w_k = 1, \quad w_k \geq 0 \tag{8}$$

At sampling time, one substrategy is chosen at random with probability w_k, and a direction sample is then generated from the chosen strategy. The two-stage randomness—first choosing a strategy, then choosing a direction—ensures that the overall sampling density at each direction is exactly the weighted average of the individual PDFs. The corresponding Monte Carlo estimator is:

$$\langle L_o \rangle = L_e + \frac{1}{N} \sum_{k=1}^{N} \frac{f_r(\omega_k) \cdot L_i(\omega_k) \cdot \cos\theta_k}{p_{\text{mix}}(\omega_k)} \tag{9}$$

The critical question is the choice of the weights $\{w_k\}$. The simplest scheme is equal-weight mixture—all w_k are equal. For M strategies, $w_k = 1/M$. Equal-weight mixture is simple and intuitive, but it does not differentiate between the quality of the strategies: in a given direction, if strategy A has a much higher PDF value than strategy B, equal-weight mixture gives them the same weight, diluting the advantage of the higher-PDF strategy. The estimator with equal weights remains unbiased, but its variance is larger than that of the optimal weighting scheme.

The balance heuristic is the optimal weight proven by Veach to minimize an upper bound on variance. Its weight function is:

$$w_k(\omega) = \frac{p_k(\omega)}{\sum_{j=1}^{M} p_j(\omega)} \tag{10}$$

Unlike equal weights, the balance heuristic's weights depend on the sampled direction ω—in any given direction, the strategy with the larger PDF value receives a higher weight. Each strategy plays a dominant role in the region where it excels (where its PDF is large) and automatically concedes in regions where others excel.

This design ensures that $p_{\text{mix}}(\omega)$ is at least of the same order as the best substrategy in every direction, keeping the worst-case f/p ratio within a bounded range.

Another advantageous property of the balance heuristic is that it does not require the substrategy PDFs to be normalizable for sampling—only the ability to evaluate PDF values is needed. This is valuable in certain sampling strategies such as those driven by measured data distributions.

1.5 Russian Roulette

The recursion of L_i in the rendering equation means that light path lengths can, in theory, extend indefinitely. Without truncation, the path tracing algorithm would never terminate. The simplest truncation method is to set a fixed maximum bounce depth $d_{\max}$ and forcibly stop when the depth reaches this limit. This approach introduces bias—the discarded deep bounces would have made a nonzero contribution to the pixel color—though the bias becomes negligible when $d_{\max}$ is sufficiently large.

Russian roulette provides an unbiased truncation scheme. A continuation probability $p_{\text{rr}} \in (0, 1]$ is set: with probability p_{rr}, the scattering path continues to be traced; with probability $1 - p_{\text{rr}}$, it terminates. If termination is chosen, the subsequent contribution of that path is taken as zero. If continuation is chosen, the path's contribution in subsequent computations is divided by p_{rr} to compensate for the expected contribution lost from terminated paths. Formally, let the radiance that would be computed recursively be L_{next}; the Russian roulette estimator is:

$$\langle L_{\text{next}}^{\text{RR}} \rangle = \begin{cases} L_{\text{next}}/p_{\text{rr}}, & \text{with probability } p_{\text{rr}} \\ 0, & \text{with probability } 1 - p_{\text{rr}} \end{cases} \tag{11}$$

The expectation of this estimator is $p_{\text{rr}} \cdot (L_{\text{next}}/p_{\text{rr}}) + (1 - p_{\text{rr}}) \cdot 0 = L_{\text{next}}$, preserving unbiasedness. The cost is increased variance—dividing by p_{rr} amplifies the fluctuation of individual samples. In practice, p_{rr} is typically taken as the maximum channel value of the current path throughput (the accumulated product of albedos), so that high-contribution paths are retained and low-contribution paths are terminated with high probability, striking a balance between increased variance and reduced computation.

2 Migration to the GPU

2.1 Advantages of the GPU

Path tracing is inherently parallel—every ray of every pixel is independent of other pixels and rays, with no data dependencies between pixels. CPU architectures are designed for low-latency serial tasks, accelerating single-thread execution through large

caches and branch prediction; GPUs are designed for high-throughput parallel tasks, simultaneously processing large volumes of data with thousands of relatively simple compute cores.

On the CPU, the path tracer is implemented as nested loops—the outer loop iterates over pixel rows, the inner loop over pixel columns, and the innermost loop generates rays for each sample and traces them recursively. Intersection testing, scattering, and sampling for each ray are executed sequentially. At an image resolution of 7680×4320 with 256 samples per pixel, over 8.5 billion rays must be traced, and the CPU's serial execution mode means rendering time is measured in hours or even days.

The GPU distributes this workload across thousands of threads executing in parallel. Each ray or each pixel is processed independently by a single thread, with all threads simultaneously running the same shader program. Ideally, when the number of threads is sufficient to cover all pixels, the rendering time for the entire image is determined primarily by the maximum depth of a single ray rather than the total ray count. Furthermore, the GPU's video memory bandwidth far exceeds the CPU's main memory bandwidth, so large numbers of texture samples and scene data accesses do not become a bottleneck as they would on the CPU.

2.2 Adaptation: From Hardware Constraints to Design Decisions

2.2.1 Instruction Execution and Branch Divergence

The programming model of the GPU differs fundamentally from that of the CPU. Directly porting CPU path tracer code to the GPU would not only fail to achieve acceleration but would not even compile. The starting point for understanding these differences lies in the GPU's instruction execution mechanism.

GPUs schedule compute resources in thread groups. In NVIDIA's terminology, 32 threads form a warp; in AMD's terminology, 64 threads form a wavefront. All threads within a warp share the same program counter and execute in a single-instruction, multiple-thread (SIMT) fashion. This means that all threads in a warp execute the same instruction in the same clock cycle, but operate on different data. When threads diverge in branching—for example, some threads hit an object while others miss—the warp must serially execute both branch paths, with threads not participating in the current branch sitting idle. This branch divergence is a primary source of GPU performance loss.

Consider triangle intersection to illustrate how this constraint affects design. The Möller-Trumbore algorithm is the standard method for ray-triangle intersection. In a CPU implementation, if the ray is parallel to the triangle plane (determinant below a threshold), the function directly returns false—this is a short-circuit evaluation, and

subsequent computation is skipped. In a GPU warp, if one of 32 rays does not satisfy this early-exit condition, all 31 rays that have already determined they could exit must also wait for that one ray to complete the entire intersection computation. Therefore, algorithm design on the GPU tends to minimize conditional branches, replacing control flow with arithmetic operations: for example, replacing `if (discriminant < 0) return false` with computing a sufficiently large t value so that subsequent distance comparisons automatically eliminate the candidate.

2.2.2 Data Layout and Indexing

Triangle intersection also reveals another constraint: data layout. On the CPU, each triangle is an independent heap object linked by pointers. The GPU does not permit this fragmented memory access pattern—threads within a warp need to access contiguous memory addresses to achieve coalesced accesses and high bandwidth. Consequently, all vertex, normal, and UV data for triangles are flattened into contiguous arrays, with each thread computing offsets based on its triangle index to perform reads. This naturally leads to the most fundamental design decision in GPU path tracing: all scene data—vertices, normals, materials, textures, BVH nodes—must be organized as flat arrays accessible via integer indices.

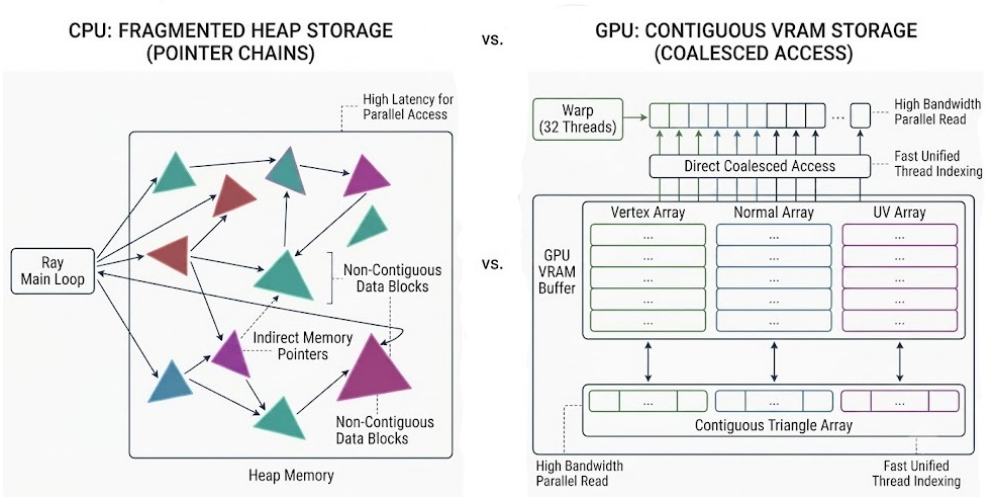

Figure 2: Comparison between pointer-linked tree data structures on the CPU and flat arrays referenced by integer indices on the GPU.

The disappearance of pointers has cascading consequences. On the CPU, BVH left and right children are referenced via pointers, materials are invoked through polymorphic interface indirect calls, and textures are dynamically loaded by file paths. None of these mechanisms work on the GPU. Acceleration structure nodes must reference children via integer offsets—internal nodes store the left child's index within its owning array, with the right child following immediately or addressed via an additional offset field. To distinguish internal nodes from leaf nodes, the highest bit is typically used as

a flag: set indicates an internal node (children to be pushed onto the stack), cleared indicates a leaf node (containing the starting index and count of a contiguous group of primitives). The material system no longer has virtual function tables: all materials are merged into a single unified structure, with a type tag field distinguishing diffuse, metal, dielectric, and other categories, and explicit branching in the shader handles each type. The same set of floating-point fields—such as roughness and refractive index—carries different physical meanings depending on the material type; this reuse keeps the storage size of each material consistent, enabling efficient array indexing.

2.2.3 Elimination of Polymorphism

Once polymorphic dynamic dispatch disappears, all branches are determined at compile time. On the CPU, an indirect jump through a material pointer costs one or two instructions; on the GPU, different pixels within the same warp may hit different material types, causing branch divergence—the warp is forced to serially execute multiple material branches, with threads not currently in the active material sitting idle. The strategy to mitigate this problem is to defer divergence until absolutely necessary: first compute the common parameters for all materials (hit point, normal, texture color), then use a single centralized branch block to dispatch to different scattering logic based on the material type.

2.2.4 Elimination of Recursion

The elimination of recursion is the most thorough code refactoring in GPU adaptation. Path tracing is logically a recursive process: starting from the camera, each time a surface is hit, a new ray is scattered, continuing until the energy decays below a threshold or the ray escapes the scene. On the CPU, this is naturally expressed as recursive function calls. GPU shading languages do not support recursion; the alternative is to unroll the recursion into a finite number of iterations—the main loop of the path tracer becomes a `for` loop with a fixed upper bound, each iteration corresponding to one bounce. Inside the loop, the ray origin and direction are updated, hit or miss is processed, and the loop continues to the next iteration. BVH traversal follows the same principle: the recursive depth-first traversal is rewritten as an explicit stack—node indices to be visited are pushed onto the stack, and the loop pops the top of the stack for AABB testing and intersection until the stack is empty or the hit distance is less than the entry distance of the current top-of-stack node. The stack size is fixed (typically 256 entries), sufficient to cover the BVH depth of the vast majority of scenes.

2.2.5 Random Number Generation

Random number generation faces another dimensional constraint. CPU random number libraries rely on operating-system entropy sources and thread-local state; on the GPU, each thread must independently generate its own random numbers without sharing global state. The solution is to initialize a random seed in each pixel-sample context—formed by mixing the pixel coordinates and sample index through a hash function—and then execute a lightweight linear congruential iteration within the thread. Since threads within the same warp independently maintain their own seed states, there is no contention or synchronization overhead. At low sample counts, stratified jittering within pixels reduces clustering noise; when sample counts are sufficiently large, purely random sampling suffices.

2.2.6 Data Transfer

The physical isolation of video memory from main memory defines the unidirectionality of data flow. Scene data—vertices, normals, material parameters, texture pixels—are uploaded to the GPU once at render startup and remain unchanged throughout the rendering process. The only parameters requiring dynamic updates are control variables that change per frame or per batch (such as the starting coordinates of the current tile, the number of accumulated samples), accomplished through writes of a small amount of uniform data. After rendering completes, the accumulated pixel colors are read back to the CPU for tone mapping and encoding output.

2.3 Architecture Design

Synthesizing the above constraints, the overall architecture of the GPU path tracer is organized around a single compute shader and a set of storage buffers. A compute shader is a GPU program that can be directly dispatched by the application and executed in parallel across a large number of threads, bypassing the vertex and pixel stages of the graphics pipeline and suited for general-purpose computation. All scene data—geometry primitive arrays, acceleration structure arrays, material arrays, light source lists, texture metadata and pixel data—together with camera parameters and control variables, are uploaded once to the GPU's storage buffers at render startup as read-only data shared by all threads. The sole read-write buffer is the output image, into which each thread writes its processed sample contributions via atomic accumulation at the corresponding pixel position.

The workgroup is the fundamental unit of GPU thread scheduling. Typically a relatively small two-dimensional workgroup size is chosen, such as 8×8 or 64 threads, allowing threads within each workgroup to exchange limited data through shared memory while the number of workgroups is sufficient to cover the entire image. Each thread

independently runs the complete path tracing loop: generating a random seed from pixel coordinates and sample index, constructing the primary ray from the camera, entering the bounce loop—upon a hit, evaluating the material and lighting and generating a scattering direction; upon a miss, querying the sky radiance and terminating; upon Russian roulette or depth-limit triggering, exiting early. There is no need for communication between threads—the computation of each light path is entirely independent.

Rendering scheduling must balance both parallel efficiency and operating-system timeout constraints. The Windows Timeout Detection and Recovery (TDR) mechanism forcibly resets the GPU driver if a single GPU operation exceeds approximately two seconds, causing rendering to fail. To avoid triggering TDR, rendering is decomposed into a large number of small-granularity dispatch units. The image is first partitioned into tiles of 512×512 pixels; each tile's total sample count is further split into multiple batches of 8 samples per batch. Each dispatch covers only one tile–batch combination, after which a synchronous wait confirms that the GPU has completed the current work before initiating the next dispatch. At a resolution of 7680×4320 with 256 samples per pixel, this yields approximately 4320 dispatches—each dispatch's workload is sufficiently small to complete within the TDR time limit, while total throughput is unaffected.

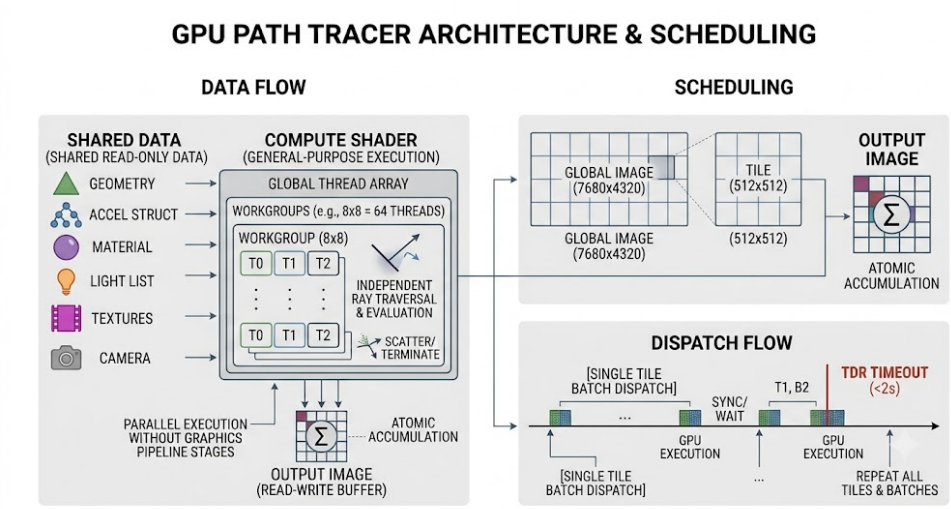

Figure 3: Schematic of the tile-batch two-level dispatch strategy. The image is divided into multiple tiles; each tile's rendering is submitted in batches, with synchronous waits between dispatches to ensure TDR safety.

2.4 The WGSL Shading Language

WGSL (WebGPU Shading Language) is the shading language defined by the WebGPU standard, providing the programming model on which the GPU path tracer's compute

shader runs. WGSL's syntax borrows Rust's keyword style (`fn`, `let`, `var`, `struct`), but its type system and resource model are entirely tailored to the constraints of GPU hardware architecture. Unlike Rust, which manages abstraction through traits and ownership, WGSL uses an attribute system to declare GPU resource bindings and scheduling parameters.

WGSL attributes are marked with the `@` prefix and placed before the item being modified. In the path tracing compute shader, the relevant attributes, in logical order, are as follows:

1. `@group(N)` and `@binding(M)` — declare the position of a global variable within a bind group. Every storage buffer variable in the shader (sphere array, triangle array, material array, output image, etc.) is located through this pair of attributes to the corresponding buffer created by the host-side wgpu API. All binding numbers `M` within the same bind group number `N` must be contiguous and non-conflicting; the layout is predefined by the host when creating the pipeline.

2. `@compute` — marks the entry function as a compute shader. Unlike vertex or fragment shaders, a compute shader does not pass through any fixed-function stage of the graphics pipeline and is launched solely through dispatch, making it suitable for general-purpose parallel computation such as path tracing.

3. `@workgroup_size(X, Y, Z)` — immediately follows `@compute`, declaring the workgroup dimensions of the shader. The GPU packs threads into groups of $(X \times Y \times Z)$ as the minimum scheduling unit. The choice of workgroup size must balance register pressure and parallelism: too large a workgroup consumes more registers, reducing occupancy; too small fails to fully utilize compute units. Path tracing commonly uses two-dimensional workgroups such as $8 \times 8 \times 1$.

4. `@builtin(global_invocation_id)` — declares a built-in input parameter that the GPU hardware automatically fills with each thread's three-dimensional coordinate in the global dispatch grid. The shader uses this to compute the pixel position for which the current thread is responsible, without needing to manually pass pixel indices.

The combination of the above attributes forms the entry skeleton of the compute shader. A typical path tracing entry function declaration is as follows:

```
@compute @workgroup_size(8, 8)
fn main(@builtin(global_invocation_id) gid: vec3<u32>) {
  let x = gid.x + u.tile_start_x;
  let y = gid.y + u.tile_start_y;
  if x >= u.tile_end_x || y >= u.tile_end_y { return; }
  // Each thread traces one pixel's path tracing loop
```

```
    var seed = hash_u32(x + y * u.image_width + sample_idx * 1234567u);
    // ...
}
```

After the entry function obtains pixel coordinates via `@builtin`, it combines them with the tile offset in the Uniform to compute the actual image coordinates. The bounds check `x >= u.tile_end_x` ensures that when the image dimensions are not divisible by the workgroup size, excess threads exit immediately without producing out-of-bounds writes.

Beyond the attribute system, WGSL has three features directly relevant to this project.

The first is the storage class. WGSL requires each global variable to explicitly declare its address space and complete its binding through `@group` and `@binding` at the declaration site. A complete declaration of scene data appears as follows:

```
@group(0) @binding(0) var<storage, read_write> output: array<vec4<f32
    >>;
@group(0) @binding(1) var<storage, read> cam: Camera;
@group(0) @binding(3) var<storage, read> spheres: array<Sphere>;
@group(0) @binding(4) var<storage, read> triangles: array<Triangle>;
@group(0) @binding(6) var<storage, read> materials: array<Material>;
@group(0) @binding(9) var<storage, read> tex_data: array<f32>;
```

`output` is the sole read-write buffer; the rest are read-only. The storage class of each variable maps directly to the GPU's hardware address space—global video memory, workgroup shared memory, or registers—and the compiler generates different types of memory access instructions accordingly. This is lower-level than Rust's ownership system: Rust's borrow checker prevents data races at compile time, whereas WGSL's storage classes directly determine the physical location and access latency of variables at the hardware level.

The second is the `ptr<function, T>` pointer type. WGSL does not support mutable references; when a function needs to modify a caller's variables, it must do so through pointer parameters. The following is the signature of a triangle intersection function that needs to simultaneously return the hit distance, UV coordinates, and interpolated normal through multiple pointer parameters:

```
fn hit_tri(
ro: vec3<f32>, rd: vec3<f32>, tmin: f32, tmax: f32, tri: Triangle,
uv_out: ptr<function, vec2<f32>>,
norm_out: ptr<function, vec3<f32>>
) -> f32 {
  // Möller-Trumbore intersection
  // Write outputs via *uv_out and *norm_out
  return t;
```

```
}
```

The caller declares local variables in function scope and passes their addresses: `hit_tri(ro, rd, tmin, tmax, tri, &uv, &norm)`. This pattern is semantically equivalent to Rust's `&mut T` but is constrained by the GPU's linear execution model—pointers cannot be null, cannot dangle, and cannot be used after the function returns; hardware naturally guarantees these safety properties.

The third is the `select()` built-in function. `select(false_val, true_val, condition)` selects between two values based on a boolean condition, and GPU hardware typically implements it via conditional move or bit-mask instructions rather than branch jumps—all threads in a warp pass through the instruction in lockstep without the branch divergence discussed in §2.2. The following comparison illustrates: handling the normal orientation upon a hit using `if-else` may introduce divergence, whereas `select()` eliminates the branch:

```
// May cause branch divergence:
if dot(rd, outward_normal) < 0.0 {
  normal = outward_normal;
} else {
  normal = -outward_normal;
}

// Branchless equivalent:
let front_face = dot(rd, outward_normal) < 0.0;
normal = select(-outward_normal, outward_normal, front_face);
```

In path tracing, frequent operations such as normal correction at hit points, conditional selection of material parameters, and sign determination of ray bounce directions can all use `select()` in place of `if-else`, converting logical branches into data selection.

WGSL's GPU-hardware-centric design means it does not provide Rust's heap allocation, recursive calls, dynamic dispatch, or external library linking, but it does provide GPU-native vector operations (`vec2/3/4<f32>` are built-in types with swizzle and component access, mapping to single SIMD instructions) and mathematical built-in functions (`dot()`, `cross()`, `normalize()`, `pow()`, `mix()`, `clamp()`, etc.).

3 Ray Intersection and Scene Acceleration

3.1 The Ray Model

A ray is described by the parametric equation $\mathbf{p}(t) = \mathbf{o} + t \cdot \mathbf{d}$ $(t > 0)$, where $\mathbf{o}$ is the ray origin and $\mathbf{d}$ is the direction vector. In path tracing, each ray carries a timestamp uniformly sampled within the shutter interval $[t_0, t_1]$, used for motion blur: during

intersection testing, object positions are interpolated according to the timestamp; different rays sample different instants, implementing Monte Carlo integration in the time dimension.

3.2 Spheres and Rectangles

The sphere is the simplest primitive for analytic intersection. Substituting the ray into $\|\mathbf{p} - \mathbf{c}\|^2 = R^2$ yields a quadratic equation $at^2 + 2bt + c = 0$, where $a = \|\mathbf{d}\|^2$, $b = \mathbf{d} \cdot (\mathbf{o} - \mathbf{c})$, $c = \|\mathbf{o} - \mathbf{c}\|^2 - R^2$. The discriminant $\Delta = b^2 - ac$ determines whether an intersection occurs; of the two roots $t = (-b \pm \sqrt{\Delta})/a$, the smallest positive root satisfying $t_{\min} < t < t_{\max}$ is taken. The normal at the hit point is given directly by $(\mathbf{p} - \mathbf{c})/R$, and UV coordinates are obtained via spherical coordinate mapping.

Rectangular primitives (XY/XZ/YZ families of planes) are solved even more directly: for an XZ rectangle, the plane equation is $y = k$, $t = (k - o_y)/d_y$, after which one checks whether the x and z coordinates lie within the boundaries. Spheres and rectangles are widely used in teaching, but neither can express arbitrary geometric shapes when rendering complex 3D models composed of triangle meshes.

3.3 Triangle Intersection

3.3.1 The Necessity of Triangle Meshes

Scenes exported by modern 3D modeling tools—whether game assets, architectural visualizations, or film-grade models—are all composed of triangle meshes. A scene of moderate complexity (such as the Narukami Shrine architecture) can contain hundreds of thousands to millions of triangles. In a CPU path tracer's pedagogical implementation, scenes are typically assembled from a small number of spheres and rectangles, which obscures the core challenge of path tracing at real data scales: each ray must perform intersection tests against all triangles in the scene to find the nearest hit point. When the triangle count reaches the millions, the intersection computation for a single ray is considerable, even with an acceleration structure.

On the GPU, the nature of this problem changes. The GPU's massive parallelism means that tens of thousands of rays can be processed simultaneously—each ray corresponds to a thread, with all threads executing intersection logic concurrently. However, as noted in §2.2, the GPU's warp execution model is highly sensitive to branch divergence. Triangle intersection contains multiple early-exit conditions and boundary checks; these branches are harmless on the CPU (and are even performance optimizations), but on the GPU they can cause serialization of threads within a warp. Therefore, triangle intersection on the GPU requires a trade-off between mathematical accuracy and warp execution efficiency—the algorithm itself is the same as on the

CPU, but implementation details are shaped by hardware constraints.

Furthermore, triangle intersection is the only intersection primitive in the GPU path tracer that can fully express arbitrary geometric shapes. Spheres and rectangles retreat to auxiliary roles (used for light source sampling and the sky, respectively), while the core scene is entirely carried by triangles. Understanding the GPU implementation of the Möller-Trumbore algorithm is the critical link for understanding how the entire rendering pipeline proceeds from ray coordinates to the final pixel color.

3.3.2 The Möller-Trumbore Algorithm

Any point on a triangle can be expressed in barycentric coordinates (β, γ) as $\mathbf{p} = (1-\beta-\gamma)\mathbf{v}_0 + \beta\mathbf{v}_1 + \gamma\mathbf{v}_2$, where $\beta, \gamma \geq 0$ and $\beta + \gamma \leq 1$. The key idea of the Möller-Trumbore algorithm is to combine the ray's parametric equation with the barycentric coordinate expression and solve for the three unknowns t, β, γ in a single step, without the need to first find the plane intersection point and then check whether it lies inside the triangle.

Let $\mathbf{e}_1 = \mathbf{v}_1 - \mathbf{v}_0$, $\mathbf{e}_2 = \mathbf{v}_2 - \mathbf{v}_0$, $\mathbf{s} = \mathbf{o} - \mathbf{v}_0$. The ray equation $\mathbf{o} + t\mathbf{d} = (1-\beta-\gamma)\mathbf{v}_0 + \beta\mathbf{v}_1 + \gamma\mathbf{v}_2$ can be rearranged as $\mathbf{o} - \mathbf{v}_0 = \beta\mathbf{e}_1 + \gamma\mathbf{e}_2 - t\mathbf{d}$, i.e., a linear system:

$$\begin{bmatrix} -\mathbf{d} & \mathbf{e}_1 & \mathbf{e}_2 \end{bmatrix} \begin{bmatrix} t \\ \beta \\ \gamma \end{bmatrix} = \mathbf{s} \tag{12}$$

Cramer's rule is applied to solve this system. The key observation is that $\det([-\mathbf{d}, \mathbf{e}_1, \mathbf{e}_2]) = \mathbf{e}_1 \cdot (\mathbf{d} \times \mathbf{e}_2)$ is equivalent to a scalar triple product—computable with `dot` and `cross`. The solutions for the three unknowns are:

$$\begin{bmatrix} t \\ \beta \\ \gamma \end{bmatrix} = \frac{1}{\mathbf{e}_1 \cdot (\mathbf{d} \times \mathbf{e}_2)} \begin{bmatrix} \mathbf{s} \cdot (\mathbf{e}_1 \times \mathbf{e}_2) \\ \mathbf{d} \cdot (\mathbf{e}_2 \times \mathbf{s}) \\ \mathbf{s} \cdot (\mathbf{e}_1 \times \mathbf{d}) \end{bmatrix} \tag{13}$$

Let $\mathbf{h} = \mathbf{d} \times \mathbf{e}_2$ to reuse the intermediate result; then the denominator $a = \mathbf{e}_1 \cdot \mathbf{h}$. If $|a| < \varepsilon$ (typically $\varepsilon = 10^{-12}$), the ray is parallel to the triangle plane and no intersection is possible. Let $f = 1/a$, $\mathbf{q} = \mathbf{s} \times \mathbf{e}_1$; then:

$$\beta = f \cdot \mathbf{s} \cdot \mathbf{h}, \quad \gamma = f \cdot \mathbf{q} \cdot \mathbf{d}, \quad t = f \cdot \mathbf{q} \cdot \mathbf{e}_2 \tag{14}$$

A hit is confirmed when $\beta \geq 0$, $\gamma \geq 0$, $\beta + \gamma \leq 1$, and $t \in [t_{\min}, t_{\max}]$. In GPU implementations, the boundary checks on β and γ are typically relaxed by a tiny amount (10^{-6}) to tolerate floating-point rounding errors—otherwise, minute cracks may appear along shared edges of adjacent triangles.

Once the hit point is determined, the normal and UV coordinates are interpolated via barycentric coordinates:

$$\mathbf{n} = (1 - \beta - \gamma)\mathbf{n}_0 + \beta\mathbf{n}_1 + \gamma\mathbf{n}_2 \tag{15}$$

$$\mathbf{uv} = (1 - \beta - \gamma)\mathbf{uv}_0 + \beta\mathbf{uv}_1 + \gamma\mathbf{uv}_2 \tag{16}$$

Phong normal interpolation gives triangle meshes a visually smooth curved-surface appearance, while UV interpolation maps the hit point to texture coordinates for subsequent texture sampling.

3.3.3 GPU Implementation Considerations

In WGSL, the implementation of the Möller-Trumbore algorithm is mathematically identical to the CPU version but has subtle differences in control flow. Taking the determinant check as an example, the CPU version directly returns false when $|a| < 10^{-12}$—a short-circuit evaluation. The GPU version also checks $|a| < 10^{-12}$, but even when parallelism is confirmed, the function returns $t_{\max}$ (a very large value) rather than exiting early via a branch. This ensures that all 32 rays in a warp pass through the entire intersection function in lockstep; no thread is left waiting because another thread exited early.

The boundary checks on the barycentric coordinates follow the same principle. The CPU version can be written as three independent `if` statements, returning false if any one is unsatisfied. The GPU version tends to combine all validity conditions into a single logical expression, or to set the t value to $t_{\max}$ to eliminate candidates that fail the conditions—the advantage being that all threads in a warp execute the same sequence of instructions with no idle waiting.

Another GPU-specific consideration is the orientation of the triangle normal. In the recursive bounces of path tracing, the actual normal direction at a hit point depends on whether the ray is incident from the front or the back face. This determination—`dot(rd, geometric_normal) < 0.0`—is inherently a branch. WGSL's `select()` function can convert the branch into data selection; as discussed in §2.2, hardware implements this via conditional move instructions, and threads in a warp do not diverge.

When the scene contains textures with alpha channels (such as petals or leaves), transparency clipping must be performed after triangle intersection. In WGSL, the alpha channel value of the texture is sampled; if it falls below a threshold (typically 0.5), the hit is treated as a miss, and the search continues for the next candidate triangle within the current BVH leaf node. This step itself is a branch, but multiple-triangle traversal within the same leaf node can amortize the divergence cost through a unified loop.

3.4 Acceleration Structure

3.4.1 GPU Traversal with an Explicit Stack

Ray-triangle intersection against every triangle in a scene of millions of triangles is unacceptably expensive. A bounding volume hierarchy (BVH) recursively partitions the set of triangles into a tree of bounding boxes, allowing each ray to test only $O(\log N)$ nodes before finding the nearest hit triangle.

GPU-side BVH traversal cannot use recursion and must rewrite the recursive depth-first traversal into explicit stack iteration (§2.2). The stack is a fixed-size `array<u32, S>`, where S is the maximum stack depth, and a stack pointer `sp` tracks the number of entries currently in the stack. The traversal proceeds as follows: the root node is pushed onto the stack; in a loop, the top-of-stack node is popped and AABB-tested; internal nodes have their left and right children sorted by hit distance and pushed onto the stack (the nearer child is pushed last, ensuring it is popped first on the next iteration); leaf nodes have their contained triangles iterated for intersection.

The choice of stack size S is an engineering trade-off. Too small a stack risks overflow during deep traversals—when a ray passes through densely packed triangle regions in the scene, the BVH traversal path may simultaneously contain dozens of pending internal nodes. If the stack is full at this point, new nodes cannot be pushed and must either be discarded or fall back to linear scanning, producing logical errors or performance degradation. Too large a stack consumes GPU register resources: each thread must maintain the entire stack array within its own register file. As the per-thread register usage increases, the number of threads the GPU can concurrently schedule on each compute unit (occupancy) decreases, reducing the parallelism available to hide memory access latency. In practice, a stack size of 256 is the validated balance point for path tracing scenes—sufficient to accommodate the number of pending nodes when the BVH depth reaches nearly a hundred levels, while keeping register pressure manageable.

The number of iterations in the traversal must have an upper bound to prevent infinite loops. In theory, a finite scene, a finite number of BVH nodes, and the stack operation of popping one node per iteration guarantee termination. However, floating-point rounding errors may cause AABB tests to produce inconsistent results (the same node being repeatedly pushed onto the stack), or logic errors in the stack operations may prevent `sp` from ever returning to zero. The iteration cap (typically set at 2000) is a defensive measure: if reached, the loop is forcibly exited and the current ray is marked as a miss.

3.4.2 Splitting Strategies: Object Median vs. Spatial Median

The core operation of BVH construction is, given a node containing N triangles, choosing a splitting plane to partition them into left and right child nodes. The quality of the split directly determines traversal performance—a good split makes the left and right bounding boxes tight and non-overlapping; a poor split produces bounding boxes with large overlap, forcing rays to traverse both subtrees.

The object median split is the most direct method: sort the triangles by the coordinate of their bounding-box centers along a chosen axis, and cut at the midpoint of the array, so that the left and right subtrees each contain $N/2$ triangles. This strategy guarantees balanced primitive counts in the two subtrees and a tree depth of $O(\log N)$. Its defect is that it completely ignores the spatial distribution of triangles—when triangles are unevenly distributed in space (for example, half the triangles in the scene concentrated in a small region while the other half are scattered across a wide region), the midpoint cut may split spatially adjacent triangles into different subtrees, causing large overlap between the left and right bounding boxes.

The spatial median split disregards triangle counts and cuts at the spatial midpoint along the chosen axis—triangles fall into the left or right subtree depending on which side of the midpoint their centers lie on. This strategy partitions the scene uniformly in space; the left and right bounding boxes are completely non-overlapping (the splitting plane is their boundary), but the triangle counts in the two subtrees can be extremely unbalanced. In the extreme case where most triangles happen to lie on the same side of the midpoint, one subtree contains nearly all the triangles while the other is empty, the tree depth degenerates to $O(N)$, and traversal efficiency is no better than linear scanning.

From the perspective of GPU traversal, the degeneracy of both strategies is directly linked to the explicit-stack constraints discussed in §3.4.1. The degeneracy of the spatial median split—tree depth ballooning to $O(N)$—means that a large number of pending nodes may simultaneously accumulate on the stack during traversal. With a fixed stack size of 256, a degenerate tree reaching hundreds of levels deep will exhaust the stack space before the nearest hit point is found, causing nodes to be discarded or traversal to fail. Even if the stack does not overflow, a traversal path of depth $O(N)$ pushes the iteration count close to or beyond the upper limit (2000), and the ray may be forcibly marked as a miss as a result. On the other hand, the degeneracy of the object median split—large bounding-box overlap—does not increase depth but significantly increases stack width: a ray must track both the left and right subtrees simultaneously, doubling the stack consumption rate and similarly increasing register pressure and reducing occupancy. The defects of the two splitting strategies are amplified by the GPU's explicit-stack constraints into visible hardware problems—stack overflow and

register overflow are no longer edge cases of algorithmic theory but real risks that can be triggered under specific scene distributions and lead to rendering errors.

3.4.3 The Surface Area Heuristic (SAH)

The surface area heuristic (SAH) unifies the above two strategies within a single cost model, selecting the optimal split position by minimizing the expected traversal cost. The core assumption of SAH is that the probability of a random ray hitting a bounding box is proportional to the box's surface area—because under a uniform isotropic distribution of rays, the larger the bounding box, the greater the probability of a ray passing through it. Given a split proposal, the expected cost of traversing that node is:

$$C_{\text{split}} = C_{\text{trav}} + \frac{\text{SA}(L)}{\text{SA}(P)} \cdot C_{\text{isect}} \cdot N_L + \frac{\text{SA}(R)}{\text{SA}(P)} \cdot C_{\text{isect}} \cdot N_R \tag{17}$$

where $\text{SA}(\cdot)$ denotes the surface area of the bounding box, N_L, N_R are the triangle counts in the left and right subtrees, C_{trav} is the fixed cost of one AABB test, and C_{isect} is the fixed cost of one triangle intersection. Intuitively, the larger a child node's surface area, the more likely a random ray will hit it—if a large node contains many triangles, those triangles have a high probability of being tested, incurring high cost; if a large node contains almost no triangles, the ray is likely to waste an AABB test.

SAH does not preset a split position but instead searches along the chosen axis. In practice, this is approximated via binning: K equally spaced bins (typically $K = 16$) are placed along the split axis, and triangles are assigned to bins by their centers; prefix scans and suffix scans compute the cumulative bounding boxes and triangle counts for all possible split positions in $O(K)$ time; the $K - 1$ candidate splits are iterated and the one with the lowest cost is selected. The complexity of this binned SAH is $O(KN)$, far lower than the $O(N^2)$ of trying every triangle as a split point.

The termination condition of SAH is also given by the cost function: if the cost of the optimal split is no less than the cost of directly making the node a leaf and intersecting all its triangles ($C_{\text{isect}} \cdot N$), then further splitting yields no benefit and the current node becomes a leaf. Leaf nodes typically have a maximum triangle count (e.g., 8) to bound the traversal cost of a single leaf.

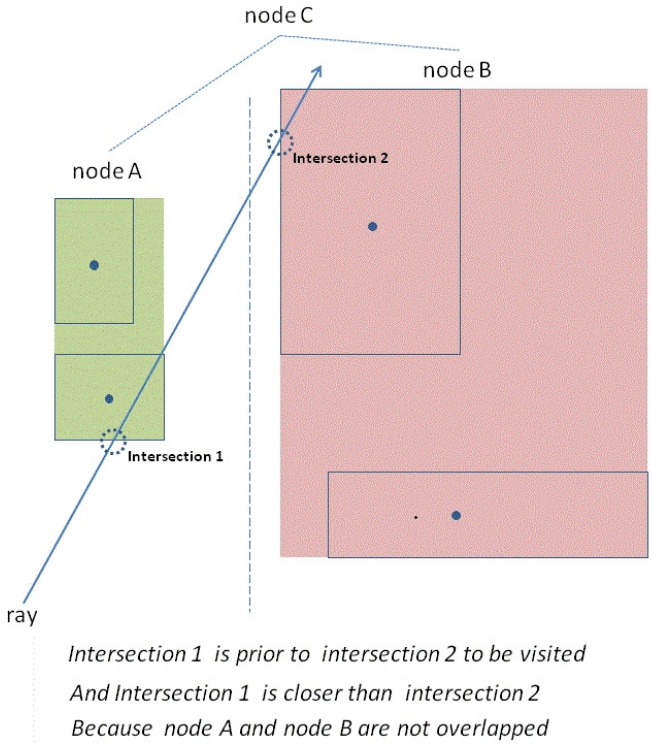

Figure 4: Comparison of three BVH splitting strategies: object median split guarantees balanced subtrees but bounding boxes may overlap; spatial median split guarantees non-overlapping bounding boxes but subtrees may be unbalanced; SAH uses a cost function to find the optimal split point between the two.

The progression from object median to spatial median and then to SAH is a journey of replacing heuristic rules of thumb with a geometric probability model. SAH finds a mathematically provable local optimum between the two—it neither ignores spatial distribution like object median, nor ignores triangle density like spatial median.

3.5 Model Loading

The OBJ format is a text format widely supported by 3D modeling tools. The loader parses vertices (`v x y z`), texture coordinates (`vt u v`, with the V-coordinate flipped to match image coordinate systems), normals (`vn x y z`), and face data (`f v1/vt1/vn1 v2/vt2/vn2 ...`). Faces support the three-component format and negative-index relative addressing (-1 refers to the most recently added vertex). Polygonal faces (quads and higher) are triangulated via fan triangulation—taking the first vertex as the anchor and sliding over subsequent vertex pairs to form a triangle fan. Materials are loaded via MTL files associated through `mtllib`, with `usemtl` switching the current material index between faces.

4 Physical Behavior of Light

The material system defines the distribution of scattering directions of light at surfaces, but the propagation of light in those directions itself obeys deeper physical laws—reflection, refraction, absorption, and subsurface scattering. These laws are independent of the specific form of the BRDF and constitute the common foundation of all material types.

4.1 Specular Reflection and Snell's Refraction

Specular reflection is the most fundamental form of light interaction with smooth surfaces. The specular reflection direction of an incident direction $\mathbf{v}$ about a normal $\mathbf{n}$ is:

$$\mathbf{r} = \mathbf{v} - 2(\mathbf{v} \cdot \mathbf{n})\mathbf{n} \tag{18}$$

This is a direct consequence of vector geometry: projecting the incident vector onto the normal direction and subtracting twice the normal component yields the reflection direction. The WGSL implementation is a single inline vector operation:

```
fn reflect(v: vec3<f32>, n: vec3<f32>) -> vec3<f32> {
  return v - 2.0 * dot(v, n) * n;
}
```

Refraction obeys Snell's law $\eta_1 \sin\theta_1 = \eta_2 \sin\theta_2$. In path tracing, the vector form of the refraction direction is more practical:

$$\mathbf{t} = \eta\mathbf{v} + \left(\eta\cos\theta_1 - \sqrt{1 - \eta^2(1-\cos^2\theta_1)}\right)\mathbf{n} \tag{19}$$

where $\eta = \eta_1/\eta_2$ and $\cos\theta_1 = -\mathbf{v}\cdot\mathbf{n}$. The expression $\eta^2(1-\cos^2\theta_1)$ inside the square root is $\eta^2\sin^2\theta_1$—when it exceeds 1, Snell's law has no real solution, corresponding to the physical condition of total internal reflection: the incident angle exceeds the critical angle $\theta_c = \arcsin(\eta_2/\eta_1)$. The WGSL implementation exploits this mathematical correspondence, using `abs` to guard the small negative values inside the square root caused by floating-point error:

```
fn refract2(uv: vec3<f32>, n: vec3<f32>, eta: f32) -> vec3<f32> {
  let ct = min(dot(-uv, n), 1.0);
  let rp = eta * (uv + ct * n);
  return rp - sqrt(abs(1.0 - dot(rp, rp))) * n;
}
```

The value of the refractive index ratio `eta` depends on whether the ray is incident from the front or back face: front-face incidence gives $\eta = 1.0/\text{ref\_idx}$ (light entering the medium from air), while back-face incidence gives $\eta = \text{ref\_idx}$ (light returning from

the medium to air). This determination is provided by the `front_face` boolean in the hit record. The total internal reflection check `eta * sin_theta > 1.0` is replaced in the code by a check on `dot(rp, rp)`; the two are mathematically equivalent but the latter requires fewer operations.

4.2 The Schlick Fresnel Approximation

Once the physical directions of reflection and refraction are determined, the next question is: how much energy is carried away by the reflection direction, and how much by the refraction direction? The full Fresnel equations involve separate computations for the s-polarized and p-polarized components—requiring averaging for unpolarized light and depending on the complex refractive index. Schlick (1994) provided a concise approximation requiring only five multiplications and four additions, with an error of less than 1% at all incident angles:

$$R(\theta) = R_0 + (1 - R_0)(1 - \cos\theta)^5 \tag{20}$$

The normal-incidence reflectance $R_0 = ((\eta_1 - \eta_2)/(\eta_1 + \eta_2))^2$ is determined solely by the refractive indices of the two media. The physical intuition behind the Schlick approximation is that the reflectance transitions smoothly and monotonically from R_0 at normal incidence to 1 at grazing incidence ($\theta \to 90°$)—any surface becomes a perfect mirror at edge-on viewing angles.

In path tracing, the Fresnel reflectance is used not as a multiplicative factor within the BRDF, but as the decision probability for Russian roulette: a uniform random number is generated; if it is less than $R(\theta)$, the reflection path is taken; otherwise, the refraction path is taken. This ensures that at any given hit point, each ray traces only one path (reflection or refraction) rather than tracing both simultaneously with weighted averaging—the latter would cause an exponential explosion in ray count in the recursive structure, whereas the former preserves the unbiasedness of a single path through probabilistic selection.

The complete decision logic for dielectric materials in WGSL is:

```
var eta = 1.0 / mat.ref_idx;
if !hit.front_face { eta = mat.ref_idx; }
let ct = min(dot(-unit_dir, normal), 1.0);
let st = sqrt(1.0 - ct * ct);

if eta * st > 1.0 {
  // Total internal reflection - force reflect
  dir = reflect(unit_dir, normal);
} else {
  let r0 = (1.0 - eta) / (1.0 + eta);
```

```
    let r0s = r0 * r0;
    let fresnel = r0s + (1.0 - r0s) * pow(1.0 - ct, 5.0);
    if rand(seed) < fresnel {
      dir = reflect(unit_dir, normal);
    } else {
      dir = refract2(unit_dir, normal, eta);
    }
  }
```

This code embodies the complete translation from theory to practice: the Schlick formula turns from a mathematical expression into three lines of WGSL statements; the mathematical condition for total internal reflection $\eta \sin\theta_1 > 1$ becomes the directly computable `eta * st > 1.0`; and the Fresnel probability determines the ray's fate through `rand(seed) < fresnel` in a Russian roulette decision.

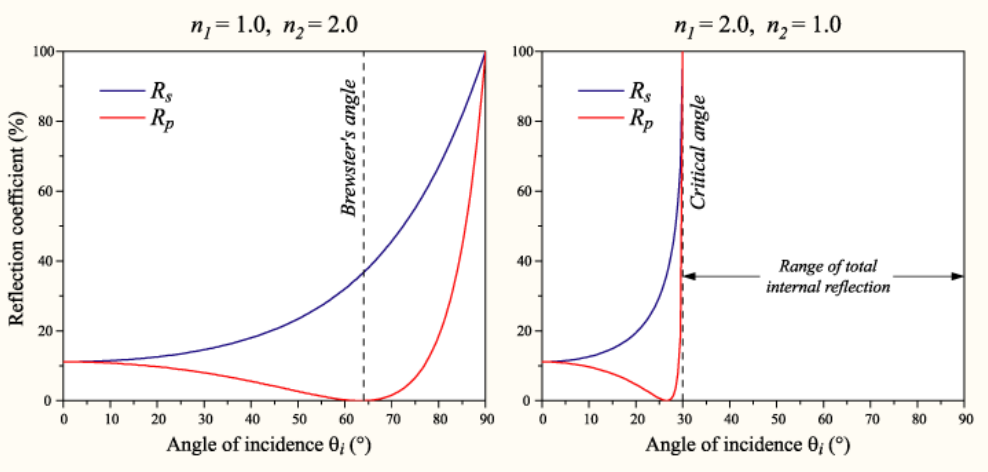

Figure 5: Relationship between reflectance and incident angle in the Fresnel effect.

4.2.1 Wave Normal Perturbation

The computation of Fresnel reflection and refraction directions depends on the surface normal. For flat or smoothly curved surfaces, the normal is directly given by the geometry. For surfaces with microscopic undulating detail, such as water, using the geometric normal would produce reflection and refraction directions that are too regular, lacking the irregular sparkle and refractive distortion characteristic of natural water. Wave normal perturbation modifies the effective normal before the Fresnel computation, causing the specular reflection and refraction directions to undergo continuous spatial and temporal variations.

The perturbation is generated by superimposing multiple layers of sinusoidal waves. Each layer has a different frequency, amplitude, and phase offset, deflecting the normal along the tangent and bitangent directions. The WGSL implementation is:

```
fn water_wave_normal(p: vec3<f32>, base_n: vec3<f32>) -> vec3<f32> {
```

```
    let t = normalize(cross(av, base_n));    // Tangent direction in
        tangent plane
    let b = cross(base_n, t);                // Bitangent direction in
        tangent plane
    let coord = p * 0.4;
    let w1 = sin(coord.x*3.7 + coord.z*2.3)
    * cos(coord.z*4.1 - coord.x*1.7) * 0.022;
    let w2 = sin(coord.x*7.1 + coord.z*5.3 + 0.7)
    * cos(coord.z*8.2 - coord.x*3.9 + 1.1) * 0.009;
    let w3 = sin(coord.x*13.4 + coord.z*9.7 + 1.3)
    * cos(coord.z*11.8 - coord.x*7.2 + 0.4) * 0.003;
    let perturb = t * (w1 + w2 + w3) + b * (w1*0.7 + w2*0.8 + w3*0.9);
    return normalize(base_n + perturb);
}
```

The three wave layers have frequencies increasing from approximately 4 rad/m to 13 rad/m, with amplitudes decreasing from 0.022 to 0.003—the low-frequency waves contribute the bulk undulation, while the high-frequency waves contribute the fine surface texture. The perturbation is applied asymmetrically along the tangent and bitangent directions (bitangent coefficients of 0.7, 0.8, and 0.9 respectively), so that the waves exhibit different visual variations from different viewing directions, avoiding isotropic repetition. The perturbed normal replaces the original geometric normal and enters the Fresnel reflection/refraction computation, so that the reflection and refraction directions acquire continuous variations consistent with the water surface undulation.

4.3 Beer-Lambert Absorption and Medium Boundary Tracking

When light propagates through a participating medium—whether that medium is water, colored glass, or translucent material—the radiance decays exponentially along the path. The Beer-Lambert law gives the transmittance after propagating a distance d through a homogeneous medium:

$$T(d) = e^{-\sigma_a \cdot d} \tag{21}$$

The extinction coefficient σ_a varies with wavelength and is thus represented as an RGB three-component vector—different colors of light are absorbed at different rates. This differential absorption is the physical origin of non-neutral coloration in participating media: wavelengths with high absorption are rapidly filtered out, while those with low absorption propagate farther in the medium. In path tracing, Beer absorption accumulates along the entire light path and is not associated with any

single bounce. In the main loop of the path tracer, whenever a ray passes through the medium between two bounces, the absorption over that distance is applied to the throughput:

```
if inside_medium {
  if hit.t < INF {
    thr *= exp(-sigma_a * min(hit.t, MAX_DIST));
  } else {
    col += thr * background * exp(-sigma_a * ESCAPE_DIST);
    break;
  }
}
inside_medium = false;
```

The `thr` (throughput) variable tracks the cumulative attenuation coefficient along the light path. The distance clamp `MAX_DIST` (e.g., 20 meters) prevents floating-point underflow of $\exp(-\sigma \cdot d)$ at extremely large distances. When a ray misses all objects and escapes the scene, an additional absorption over a fixed escape distance must still be multiplied in to simulate the attenuation of skylight by the medium.

Tracking of medium boundaries requires determining whether the ray is currently inside or outside the medium. This determination is not provided independently by the material properties of the hit point—it depends on the ray's direction of origin, since a ray may start from inside the medium and, after hitting another surface from the inside, remain within the medium. The tracking method maintains a boolean state in the path tracing loop: when a ray enters the medium from the outside via refraction, it is set to true; when it leaves the medium from the inside via refraction, it is set to false; when it is reflected back inside via total internal reflection, it remains true. This state is carried across bounces, allowing each subsequent ray to correctly accumulate Beer absorption along its path.

4.3.1 Atmospheric Fog Attenuation

Similar to Beer absorption inside water, light propagating through air also experiences distance-dependent attenuation—aerosols and particulates in the atmosphere scatter and absorb visible light, reducing the contrast of distant objects and shifting their color toward the background. Unlike the per-wavelength absorption of water, the extinction effect of atmospheric fog is approximately wavelength-independent in the visible band—the color of fog comes mainly from scattered ambient light rather than selective absorption by the medium itself—so a scalar extinction coefficient is used instead of an RGB vector.

Atmospheric fog attenuation accumulates along every light path, sharing the same exponential attenuation form $T(d) = e^{-\sigma \cdot d}$ as Beer absorption in water, but the ex-

tinction coefficient σ is a scalar (typically on the order of 10^{-4} to $10^{-3}\,\mathrm{m}^{-1}$), and the attenuation occurs on all ray paths, not only inside water. In the main loop of the path tracer, the distance between each pair of bounces applies attenuation to the current throughput:

```
let fog_trans = exp(-hit.t * FOG_DENSITY);
thr = thr * fog_trans;
```

When a ray misses all objects and escapes to the sky, the atmospheric scattering at large distances blends the sky color with the fog color. The blend weight is given by the complement of the exponential attenuation ($1 - e^{-\sigma d}$); at large distances (d large), the fog color weight approaches 1, and the object is completely obscured by fog:

```
let fog_blend = 1.0 - exp(-DIST * FOG_DENSITY);
col = col + thr * mix(sky_color, FOG_COLOR, fog_blend);
```

Unlike the medium boundary tracking of §4.3, atmospheric fog attenuation requires no state tracking—it is present on all ray paths, regardless of whether the ray is bouncing off a surface, escaping to the sky, or passing through water. It is an ever-present background attenuation term in the main loop of the path tracer.

4.4 Subsurface Scattering — Thin-Slab Random Walk

4.4.1 Physical Mechanism

Subsurface scattering (SSS) is a common optical characteristic of translucent materials such as skin, wax, jade, and leaves. Light penetrates the surface of such materials, undergoes multiple scattering events inside the medium, and eventually escapes from a position different from the point of entry. This process gives translucent objects a soft volumetric light sensation in appearance: edge glow under backlighting, with surface reflection and internal scattering jointly determining the final color.

Mathematically, SSS requires a generalization of the rendering equation. The standard BRDF assumes that light enters and exits at the same point—the point of incidence and the point of exit coincide. In subsurface scattering, however, light can exit from any point within a region around the point of incidence; the incident radiance is linked to the outgoing radiance through the internal transport paths of the medium. The BSSRDF (bidirectional subsurface scattering reflectance distribution function) generalizes the BRDF to an 8-dimensional function $S(\mathbf{p}_i, \omega_i, \mathbf{p}_o, \omega_o)$—indexed respectively by the surface positions of the incidence and exit points (each parameterized by two dimensions) and the incident and outgoing directions (each parameterized by two dimensions of solid angle). The outgoing radiance thus becomes a double integral

over the incident surface area and the incident directional hemisphere:

$$L_o(\mathbf{p}_o, \omega_o) = \int_A \int_{H^2} S(\mathbf{p}_i, \omega_i, \mathbf{p}_o, \omega_o) \cdot L_i(\mathbf{p}_i, \omega_i) \cdot \cos\theta_i \, d\omega_i \, dA_i \tag{22}$$

The value of the BSSRDF is determined by the solution of the radiative transfer equation inside the medium—a differential equation describing how light simultaneously undergoes absorption and scattering in a participating medium. For homogeneous media, this can be solved through the diffusion approximation (treating multiple scattering as a diffusion process) or through Monte Carlo volumetric path tracing. Importance sampling and evaluating the full BSSRDF in per-pixel path tracing is prohibitively expensive—every surface hit would require sampling a surface region around the point of incidence and tracing the full path of photons inside the medium, with a computational cost far exceeding that of standard surface BRDF evaluation.

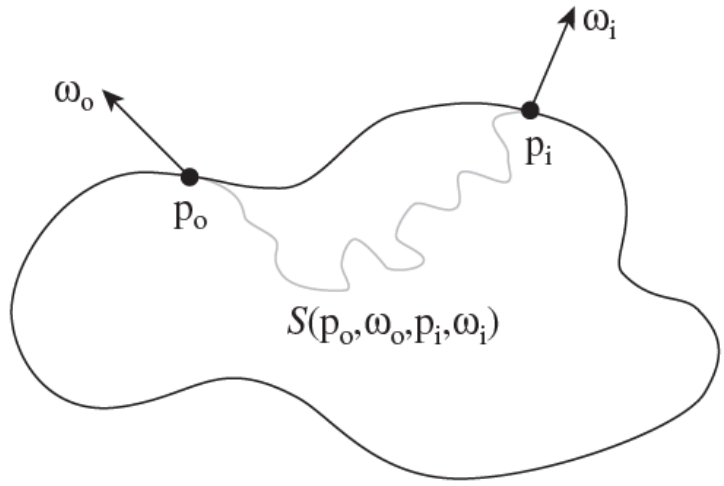

Figure 6: The BSSRDF illustrated as the function S in the rendering equation generalization.

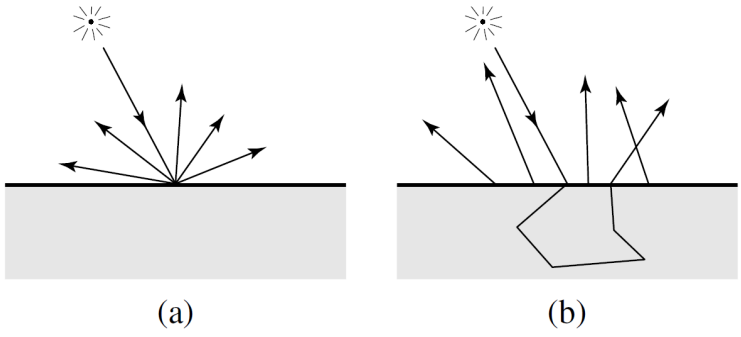

Figure 7: Subsurface scattering behavior inside a medium is considerably complex, exhibiting diverse optical manifestations.

The thin-slab random walk is a class of methods that simplify the BSSRDF into a

Monte Carlo simulation. It assumes the medium is a homogeneous flat slab of finite thickness, with photons entering the medium interior in the direction opposite to the surface normal and performing a discrete random walk inside—each step samples a free path to determine how far to travel, samples a phase function to determine which direction to turn, and has a certain probability of being absorbed by the medium. When a photon walks back to the surface (depth < 0) or penetrates to the back face (depth $>$ thickness), the walk terminates and the photon escapes from the corresponding surface. The path tracing framework naturally supports this simulation: after Fresnel refraction brings light into the medium interior, an inner loop replaces the surface BRDF scattering, and at the end of the loop, tracing continues from the escape surface.

4.4.2 Random Walk Algorithm

The core of the algorithm is an iterative loop inside the medium. At the hit point, the inward direction of the medium is taken as the opposite of the surface normal (`inward_n = -n`), establishing the thickness reference. Each iteration performs five operations:

```
let sigma_t = sigma_s + sigma_a;                        // Extinction
    coefficient
let albedo_s = sigma_s / sigma_t;                       // Single-
    scattering albedo
let sigma_t_avg = (sigma_t.r + sigma_t.g + sigma_t.b) / 3.0;

for (var step: u32 = 0u; step < MAX_STEPS; step++) {
  // 1. Sample free path (inverse CDF of exponential distribution)
  let d_free = -log(max(rand(seed), EPS)) / sigma_t_avg;
  pos += d_free * dir;
  let depth = dot(pos - hit_point, inward_n);

  // 2. Check escape
  if depth < 0.0 {
    pos -= depth * inward_n;                        // Front-face
        escape
    escaped_front = true; break;
  }
  if depth > thickness {
    pos -= (depth - thickness) * inward_n;          // Back-face escape
    escaped_back = true; break;
  }

  // 3. Accumulate absorption and scattering
  thr *= exp(-sigma_a * d_free) * albedo_s;
```

```
        // 4. Russian roulette termination
        let survival = max(thr.r, max(thr.g, thr.b));
        if survival < RR_THRESHOLD || rand(seed) > survival { break; }
        thr /= survival;                                    // Unbiased
            compensation

        // 5. Sample new scattering direction
        dir = sample_hg(dir, g, seed);
    }
```

Free path sampling $d = -\ln(\xi)/\sigma_t$ follows the inverse CDF of the exponential distribution. The single-scattering albedo $\alpha = \sigma_s/\sigma_t$ is the probability that a photon is not absorbed in a single scattering event—the closer α is to 1, the more scattering dominates, and the brighter the medium (like milk); the lower α, the stronger the absorption, and the darker the medium (like coffee). Russian roulette uses the maximum channel value of the current throughput as the continuation probability; when the energy has decayed to an extremely low level, the walk is terminated with high probability, and dividing by the probability preserves the unbiasedness of the estimate. The thickness parameter determines the ease of transmission: thin media (like petals or paper) allow many photons to escape from the back face, producing a strong transmitted-light effect; thick media (like paraffin or marble) are nearly impenetrable to back-side light, and the SSS contribution comes mainly from the diffuse light escaping from the front face.

4.4.3 Anisotropic Phase Functions

Free path sampling determines the distance the photon travels, while the deflection of the scattering direction is described by a phase function. The simplest phase function is isotropic scattering, $p(\omega) = 1/(4\pi)$—photons are deflected with equal probability in all spatial directions, independent of the incident direction. Isotropic scattering is mathematically simple and highly efficient for Monte Carlo sampling (requiring only uniform sampling of directions on the unit sphere), but it is an inaccurate approximation for most real media. The tiny water droplets in clouds and fog, the scattering particles in translucent materials, and the cellular structures in biological tissues all exhibit significant directional preferences—photons tend to continue scattering at angles close to the original incident direction rather than turning randomly.

Rayleigh scattering is the earliest analytic theory of anisotropic scattering, describing the interaction of light with spherical particles much smaller than the wavelength (such as nitrogen and oxygen molecules in the atmosphere). The Rayleigh scattering phase function is $p(\cos\theta) = \frac{3}{16\pi}(1 + \cos^2\theta)$, exhibiting a symmetric dumbbell-shaped distribution—forward and backward scattering probabilities are equal, with side scat-

tering (90°) being the lowest. Its total scattering cross section is proportional to λ^{-4}, explaining why the sky is blue (short-wavelength blue light is scattered approximately 10 times more strongly than long-wavelength red light) and why sunsets are red. However, Rayleigh theory applies only when the particle radius is much smaller than the wavelength ($r < \lambda/10$); for scattering particles in most participating media—with diameters comparable to or larger than the wavelength—the more general Mie scattering theory must be used.

Mie scattering is the exact solution of Maxwell's equations for spherical particles, applicable to arbitrary ratios of particle size to wavelength. Unlike Rayleigh scattering, Mie scattering exhibits strong forward dominance—the larger the particle, the more concentrated the forward scattering. Its phase function is an infinite series involving Legendre polynomials, which has no analytic sampleability in Monte Carlo sampling, and accuracy control of series truncation is complex. Therefore, in practical rendering, the exact Mie scattering phase function is rarely used directly; it is replaced by analytic approximation models that retain its core feature—a single parameter controlling the forward/backward deflection tendency.

The Henyey-Greenstein (HG) phase function was designed precisely for this purpose:

$$p(\cos\theta) = \frac{1}{4\pi} \cdot \frac{1 - g^2}{(1 + g^2 - 2g\cos\theta)^{3/2}} \tag{23}$$

The anisotropy parameter $g \in [-1, 1]$ directly corresponds to the asymmetry factor of Mie scattering (the mean of the cosine of the scattering angle)—$g = \langle\cos\theta\rangle$. When $g = 0$, it degenerates to isotropic scattering; when g approaches $+1$, the phase function has a sharp peak at $\theta = 0$ (forward), corresponding to the Mie scattering characteristics of large particles; when g approaches -1, backscattering dominates. The HG phase function cannot reproduce the symmetric dumbbell shape of Rayleigh scattering (it cannot simultaneously peak in both the forward and backward directions), but it approximates the angular distribution of most Mie scattering scenarios well through first-moment matching.

The key advantage of the HG phase function in Monte Carlo methods is that its inverse CDF has a closed-form expression:

$$\cos\theta = \frac{1}{2g}\left[1 + g^2 - \left(\frac{1 - g^2}{1 - g + 2g\xi}\right)^2\right] \tag{24}$$

This makes the cost of sampling a direction from the HG distribution comparable to that of isotropic sampling—requiring only one uniform random number and a handful of multiply-add operations. The implementation must handle the degenerate case when $g \approx 0$: when $|g| < 10^{-3}$, the parameter g appears in the denominator of the formula and floating-point error is amplified; at this point, the code falls back directly

to isotropic sampling $\cos\theta = 1 - 2\xi$. After sampling (θ, ϕ), the scattering direction in world space is constructed in the local coordinate system of the incident direction ω_o:

```
fn sample_hg(wo: vec3<f32>, g: f32, seed: ptr<function, u32>) -> vec3
    <f32> {
  let xi = rand(seed);
  var cos_theta: f32;
  if abs(g) < 0.001 {
    cos_theta = 1.0 - 2.0 * xi;                    // Degenerate to
       isotropic
  } else {
    let g2 = g * g;
    let term = (1.0 - g2) / (1.0 - g + 2.0 * g * xi);
    cos_theta = (1.0 + g2 - term * term) / (2.0 * max(g, 0.001));
  }
  let sin_theta = sqrt(max(1.0 - cos_theta * cos_theta, 0.0));
  let phi = 2.0 * PI * rand(seed);
  let av = select(vec3<f32>(0,1,0), vec3<f32>(1,0,0), abs(wo.y) >
     0.999);
  let t = normalize(cross(av, wo));
  let b = cross(wo, t);
  return wo * cos_theta + t * sin_theta * cos(phi) + b * sin_theta *
     sin(phi);
}
```

By adjusting the four parameters σ_a (absorption coefficient), σ_s (scattering coefficient), g (anisotropy), and thickness, the thin-slab random walk can cover a broad range of material classes: from highly forward-scattering thin translucent materials ($g \approx 0.7$–0.9) to isotropic thick media ($g \approx 0$) to backscattering-dominated materials ($g < 0$).

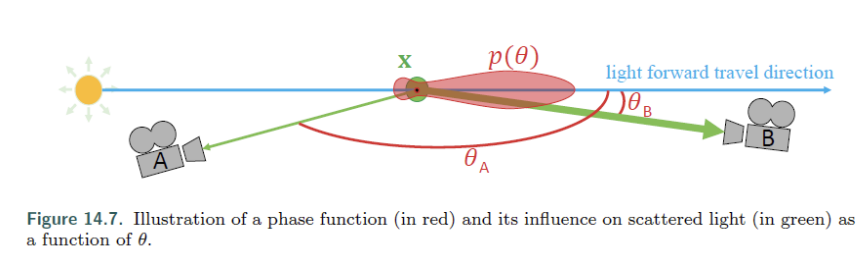

Figure 14.7. Illustration of a phase function (in red) and its influence on scattered light (in green) as a function of θ.

Figure 8: Schematic of anisotropic phase functions: isotropic vs. forward-scattering vs. backward-scattering patterns.

4.5 Microsurface Models and Normal Perturbation

4.5.1 Microsurface Models

The microsurface model is the standard framework in computer graphics for describing the scattering behavior of non-smooth surfaces, proposed by Cook and Torrance in 1981. Its fundamental assumption is that the macroscopic surface is composed, at the microscopic scale, of a large collection of randomly oriented specularly reflecting facets (microfacets). Each microfacet is itself optically smooth—incident light undergoes perfect specular reflection or Fresnel refraction on it—but because the normal directions of the facets are randomly distributed around the macroscopic geometric normal, what is observed at the macro scale is a blurred reflection lobe rather than a single specular direction.

The complete form of the microsurface model consists of three components. The normal distribution function (NDF) $D(\mathbf{m})$ describes the probability density of microfacet normals $\mathbf{m}$ around the macroscopic normal $\mathbf{n}$—commonly used specific forms include the Beckmann distribution and the Trowbridge-Reitz (GGX) distribution, all sharing the characteristic that the distribution width is controlled by a roughness parameter α. The geometry attenuation term (shadowing-masking term) $G(\omega_i, \omega_o, \mathbf{m})$ describes occlusion between microfacets: at grazing angles, some facets are blocked by adjacent protruding facets, reducing the effective reflecting area. The Fresnel term $F(\omega_i, \mathbf{m})$ is given by the Schlick approximation of §4.2, describing the reflection/refraction energy partitioning on a single microfacet. The product of the three gives the specular reflection component of the microfacet BRDF:

$$f_r(\omega_i, \omega_o) = \frac{D(\mathbf{h})\, G(\omega_i, \omega_o)\, F(\omega_i, \mathbf{h})}{4(\mathbf{n} \cdot \omega_i)(\mathbf{n} \cdot \omega_o)} \tag{25}$$

where $\mathbf{h} = (\omega_i + \omega_o)/\|\omega_i + \omega_o\|$ is the half-angle vector—the microfacet normal direction that exactly reflects the incident light into the outgoing direction.

In path tracing, importance sampling of the microfacet BRDF is typically designed around the NDF: the half-angle vector $\mathbf{h}$ is sampled from the distribution $D(\mathbf{h})$, from which the outgoing direction is derived. This scheme is efficient and low-variance at small roughnesses, but at large roughnesses it requires multiple-scattering corrections to maintain energy conservation.

4.5.2 A Simplified Fuzzy Reflection Scheme

The full microsurface model involves NDF sampling, geometry-term computation, and multiple-scattering compensation, which is relatively complex to implement. For the metal, clear-coat, and surface-fuzz materials in this project, a simplified scheme with far lower computational cost than the full microsurface model is adopted. The scat-

tering direction is taken directly as the specular reflection direction plus a uniform random vector scaled by the roughness:

$$\omega_o = \mathbf{r} + \alpha \cdot \boldsymbol{\xi} \tag{26}$$

where $\mathbf{r} = \text{reflect}(\omega_i, \mathbf{n})$ is the perfect specular reflection direction, $\boldsymbol{\xi}$ is a random vector uniformly sampled inside the unit sphere, and α is the roughness parameter. The scattering direction is subsequently normalized. If the perturbed direction points below the surface ($\omega_o \cdot \mathbf{n} \leq 0$), the ray is treated as absorbed and the path is terminated.

This simplified scheme produces visually plausible near-specular metal effects when α is small ($\alpha \approx 0.03$–0.12): the reflection highlight is slightly diffused but remains recognizable in shape. At medium roughness ($\alpha \approx 0.2$–0.3), it gives the appearance of brushed or matte metal. Compared to the full microsurface model, it lacks the tail behavior of the NDF (the long-tail distribution of GGX at high roughness produces brighter grazing-angle reflections), the geometry occlusion term (reflections from high-roughness surfaces should darken at grazing angles), and the energy compensation of multiple scattering. However, when paired with the cosine-weighted PDF evaluation of §1.3, the Monte Carlo estimate of the scattering direction remains correct—the perturbed reflection direction, after normalization by the PDF, still yields an unbiased reflection contribution.

The WGSL implementation of metal materials directly reflects this simplified scheme:

```
if mt == 1u {
  let tex_color = sample_texture(mat.tex_id, hit.uv);
  let surface_color = mat.albedo.rgb * tex_color;
  let refl = reflect(normalize(rd), hit.n);         // Specular
     reflection direction
  let fuzz_val = max(mat.fuzz, 0.0001);
  let sdir = refl + fuzz_val * rand_unit_vec(seed); // Perturb
  if dot(sdir, hit.n) <= 0.0 { break; }            // Below surface
     → absorption
  rd = sdir;
  thr = thr * surface_color;
  ro = p + hit.n * 0.002;
  continue;
}
```

The `fuzz` parameter is protected by a minimal value (`max(mat.fuzz, 0.0001)`) to prevent floating-point error from producing a perturbed direction below the surface even at zero roughness. The clear-coat material (Type 4) and the surface fuzz layer of SSS petals (Type 7) use the same form of perturbation, differing only in the roughness value—clear-coat uses 0.2–0.25 corresponding to a satin matte finish, while the fuzz

layer uses 0.65 corresponding to rough backlit scattering.

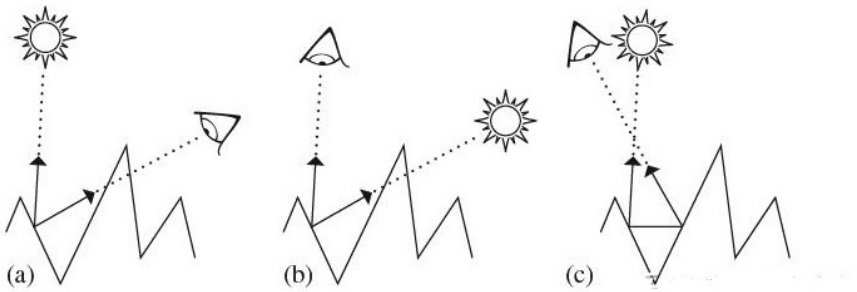

Figure 9: Several behaviors of rays under a microsurface model: blocked by obstacles or bouncing repeatedly between them.

4.5.3 Bump Mapping

Microsurface models alter the deviation of the scattering direction from the specular reflection direction, whereas bump mapping alters the normal itself—by applying a tiny perturbation to the geometric normal at each hit point, the surface acquires rich visual texture without increasing geometric complexity. Unlike wave normal perturbation (§4.2.1), the perturbation in bump mapping originates from a procedural noise field rather than analytic sine waves, making it suitable for materials with random textures such as stone, wood, and fabric.

The core operation of bump mapping is to estimate the gradient of a scalar field $f(\mathbf{p})$ (such as FBM noise) at the hit point $\mathbf{p}$, project the gradient onto the tangent plane, and use it to perturb the normal:

$$\hat{\mathbf{n}} = \text{normalize}\left(\mathbf{n} - \frac{s}{\varepsilon} \cdot \mathbf{P}_T \cdot \nabla f\right) \tag{27}$$

where $\mathbf{P}_T = \mathbf{I} - \mathbf{n}\mathbf{n}^T$ is the operator that projects the gradient onto the tangent plane—only variations within the tangent plane affect the normal direction; variations along the normal direction do not change the normal. s is the perturbation strength, and ε is the finite-difference step size. The gradient is estimated via three finite differences:

```
fn bump_normal(p: vec3<f32>, n: vec3<f32>,
field_val: f32, eps: f32, strength: f32) -> vec3<f32> {
  let dx = field_val - fbm(p - vec3<f32>(eps, 0, 0), 4u);
  let dy = field_val - fbm(p - vec3<f32>(0, eps, 0), 4u);
  let dz = field_val - fbm(p - vec3<f32>(0, 0, eps), 4u);
  let t = normalize(cross(av, n));            // Tangent
  let b = cross(n, t);                        // Bitangent
  let grad = vec3<f32>(dx, dy, dz);
  return normalize(n - (t*dot(grad,t) + b*dot(grad,b)) * (strength /
     eps));
```

```
    }
```

The finite-difference step size ε determines sensitivity to surface detail—too small a value and the gradient is drowned in noise (the scalar field has no structure at very small scales); too large a value and the spatial resolution of the gradient is insufficient. The perturbation strength s and the step size ε jointly determine the magnitude of the perturbation vector: the larger s/ε, the more the perturbed normal deviates from the original normal, and the stronger the visual bumpiness. The parameter differences between materials—the pronounced bumpiness of stone ($\varepsilon = 0.010, s = 0.04$) versus the subtle texture of wood ($\varepsilon = 0.008, s = 0.03$)—ultimately reduce to different choices of these two parameters.

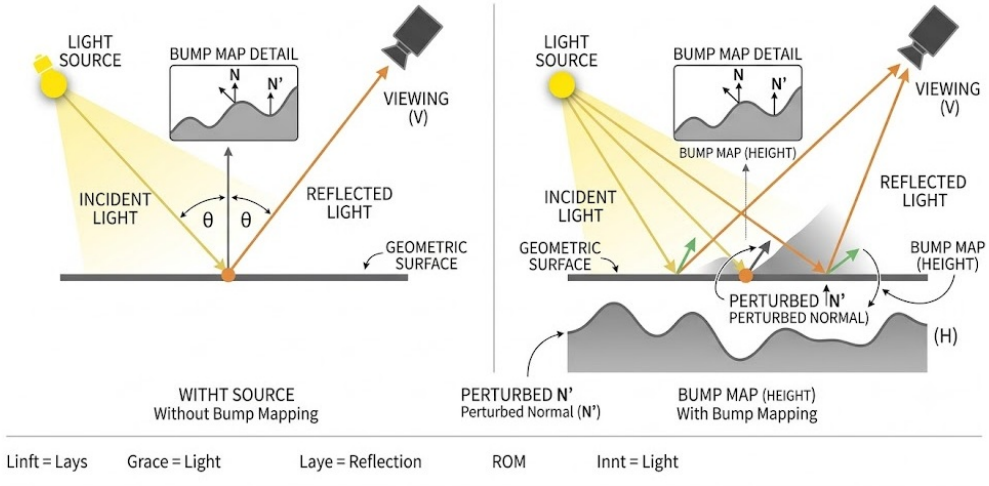

Figure 10: Schematic of bump mapping: the geometric normal is perturbed by the gradient of a scalar field to simulate surface detail without additional geometry.

5 Material System

5.1 Basic Materials

The material system of path tracing proceeds from three fundamental physical behaviors: diffuse reflection distributes incident light uniformly over all directions of the upper hemisphere; metallic reflection concentrates light near the specular direction; dielectrics apportion the light path between reflection and refraction according to Fresnel probability. The parameter spaces of these three behaviors are small—diffuse requires only albedo and an optional texture, metal requires only albedo and the roughness `fuzz`, dielectric requires only the refractive index `ref_idx`—yet they form the foundational skeleton of all complex materials. Diffuse represents the bottom-layer behavior of surface scattering: light first enters the surface and is then scattered. Metal represents pure reflective behavior. Dielectric represents reflective/refractive bifurcation

behavior. All subsequent advanced materials are superpositions of additional physical layers, texture layers, or geometric layers on top of these three basic behaviors.

5.2 Advanced Materials

5.2.1 ClearCoat

ClearCoat simulates the double-layer optical structure of a transparent protective coating over a wood substrate. Light first strikes the coating surface—a smooth dielectric film—where the Fresnel reflectance decides whether it is directly reflected, yielding the coating's highlight, or penetrates the coating into the substrate, revealing the wood's diffuse color. The coating's refractive index is similar to that of glass, with IOR 1.5 and normal-incidence reflectance $R_0 \approx 0.04$; in actual rendering, R_0 is boosted to approximately 0.06 to enhance the visual contrast of the coating highlight.

The algorithm proceeds as follows: compute the coating's Schlick $R(\theta)$; with probability $U(0,1) < R(\theta)$, take the reflection path; otherwise take the transmission path. In the reflection path, the scattering direction is the specular reflection direction plus roughness perturbation in the simplified fuzzy reflection scheme, with 70% of the reflection contribution being pure white, coming from the coating's own Fresnel reflection, and 30% coming from the base color, corresponding to the portion of light that penetrates the coating and scatters back from the substrate. In the transmission path, the base color is multiplied by an attenuation of 0.88—the combined effect of Fresnel transmittance and internal absorption in the substrate—using a darker diffuse reflection to set off the bright highlight.

```
if mt == 4u {
  let tex_color = sample_texture(mat.tex_id, hit.uv);
  let base_color = mat.albedo.rgb * tex_color;
  let r0 = (mat.ref_idx - 1.0) / (mat.ref_idx + 1.0);
  let r02 = r0 * r0 * 1.5;                          // Visual
     enhancement
  let fresnel = r02 + (1.0 - r02) * pow(1.0 - abs(dot(rd, n)), 5.0);
  if rand(seed) < fresnel {
    let refl = reflect(rd, n);
    let sdir = refl + mat.fuzz * rand_unit_vec(seed);
    thr *= mix(vec3(1.0), base_color, 0.70);        // 70% white +
       30% base
  } else {
    thr *= 0.88;
  }
}
```

ClearCoat materials spawn multiple variants in the scene: sashes and garment accessories use `fuzz` of 0.25 for a matte coating effect; dark woods use a lower base-

color albedo to make the highlight more prominent; silk-like materials can reduce `fuzz` to 0.15 and combine with anisotropic direction encoding to simulate stretched highlights along the thread direction.

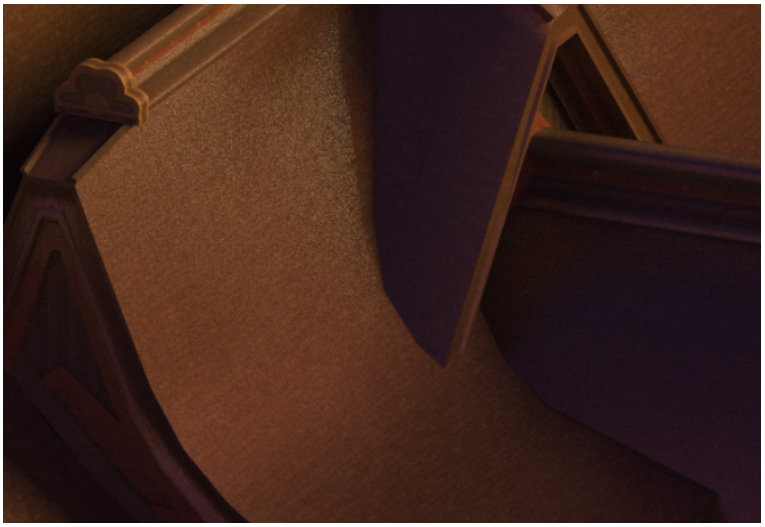

Figure 11: Visual effect of a ClearCoat double-layer varnish material.

5.2.2 Procedural Stone

The visual signature of stone arises from multi-scale surface irregularity—from centimeter-level rock-block undulations to millimeter-level grain texture to micron-level crack networks. Fractal Brownian motion (FBM) is naturally suited to describing this cross-scale self-similar texture structure.

FBM is a multi-octave superposition of a base noise function. Let $n(\mathbf{p})$ be a continuous noise function defined in three-dimensional space with range $[0, 1]$, whose spatial frequency determines the fineness of its variation. FBM takes the weighted sum of N octaves of noise: the i-th octave multiplies the input coordinate by 2^i for frequency scaling, attenuates the noise amplitude by $(1/2)^i$, and normalizes the total amplitude by dividing through the sum of all weights:

$$\text{fbm}(\mathbf{p}, N) = \frac{\sum_{i=0}^{N-1}(1/2)^i \cdot n(2^i \cdot \mathbf{p})}{\sum_{i=0}^{N-1}(1/2)^i} \tag{28}$$

Doubling the frequency at each octave causes each successive octave to produce detail twice as fine as the previous one, corresponding to the scale progression on stone from rock-block contours to grain texture to micro-cracks. Halving the amplitude at each octave embodies the universal law that high-frequency detail carries less energy than low-frequency structure in nature—the large-scale contours of rock carry the largest normal deviation, grain texture is next, and micro-cracks contribute the least to the normal but lend visual sharpness. The denominator $\sum(1/2)^i$ (whose series limit

is 2) ensures that the FBM output is always constrained to the $[0, 1]$ interval and does not diverge as more octaves are added.

Stone's `stone_field` further allocates three characteristic scales to separate FBM calls, using different input frequency ranges and independent phase offsets to avoid visual repetition between scales:

```
fn stone_field(p: vec3<f32>) -> f32 {
  let coarse = fbm(p * 3.0,  3u);                    // Coarse rock-
     block structure
  let fine   = fbm(p * 11.0, 3u) + vec3(1.7);        // Fine grain
     texture
  let cracks = fbm(p * 25.0, 2u) + vec3(3.1);        // Micro-crack
     network
  return coarse * 0.5 + fine * 0.3 + cracks * 0.2;
}
```

The three-layer weighting of $0.5 : 0.3 : 0.2$ makes the coarse structure dominate the large-scale direction of the normal perturbation. The normal perturbation uses the finite-difference gradient estimation of bump mapping. The perturbation parameters $\varepsilon = 0.010, s = 0.04$ produce visually evident bumpiness—enough to see the undulation of rock blocks but not so severe as to produce unnatural sharp edges.

Stone roughness is not a global constant but varies spatially with the FBM field value: $\alpha = \text{mix}(0.4, 0.75, f(\mathbf{p}))$. In regions of rich texture with protrusions and depressions, roughness is higher; in flat regions, it is lower. Six percent of rough specular reflection is blended into the scattering direction, simulating the faint glint of wet stone surfaces or quartz grains—this tiny blend gives stone visible bright edges at grazing angles and avoids the chalky appearance of purely diffuse reflection.

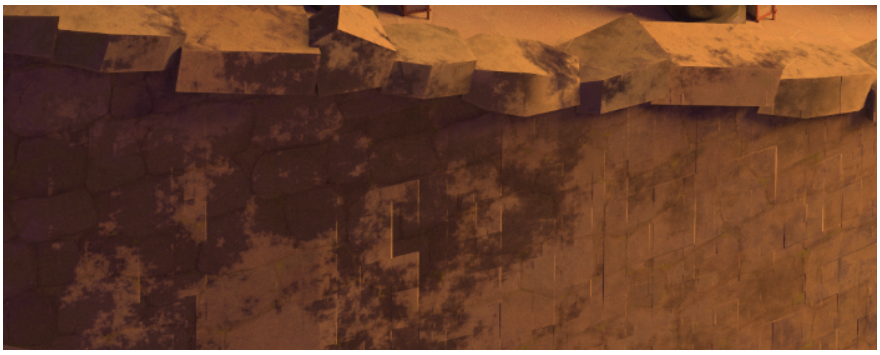

Figure 12: Visual effect of a procedural stone material.

5.2.3 Procedural Wood

The fundamental difference between wood and stone lies in its anisotropy—wood fibers are aligned along the trunk's longitudinal direction, so texture extends along the fiber direction, while cross sections perpendicular to the fibers exhibit ring-like

annual growth patterns. The optical roughness of the surface differs systematically between directions. Along the fiber direction, the surface is relatively smooth, with a concentrated microfacet normal distribution and a narrow specular reflection lobe; perpendicular to the fiber direction, growth-ring boundaries, vessel cross sections, and fiber ends constitute dense microscopic undulations, with significantly elevated roughness and a broadened reflection lobe. This anisotropic roughness manifests visually as elongated highlights stretched along the wood-grain direction—the most distinctive optical feature that sets wood apart from isotropic rough materials.

Wood roughness is not a global scalar but a scalar field defined in three-dimensional space, whose value varies with the FBM field of the wood-grain texture: positions with high texture density correspond to roughness peaks, while sparse-texture regions have lower roughness. Specifically, roughness is mapped from the output value of `wood_field` to the interval $[0.45, 0.85]$:

$$\alpha(\mathbf{p}) = \text{mix}(0.45, 0.85, \text{wood\_field}(\mathbf{p})) \tag{29}$$

This mapping synchronizes roughness with the wood-grain texture in space. In regions with dense growth rings, roughness is high and the reflection highlight is blurred more diffusely; in smooth inter-fiber regions, roughness is low and the highlight is more concentrated.

The anisotropy of roughness is achieved by compressing the world coordinate in the fiber extension direction before passing it into the FBM. Before the FBM call, the hit-point coordinate is compressed by a factor of 12 in the texture extension direction—for vertical grain, the input becomes $(x, y \times 0.12, z)$; the FBM signal varies extremely slowly along the compressed axis: noise that would change dozens of times over a 1-meter range originally now changes only a few times after compression. Roughness along the compressed axis is nearly constant, while roughness perpendicular to the compressed axis alternates periodically with the wood grain. When the scattering direction of fuzzy reflection is sampled from such a roughness field, the perturbation along the fiber direction is minimal and the reflection is sharp, while the perturbation in the perpendicular direction is larger and the reflection is blurred—the composite effect producing reflection highlights stretched along the wood-grain direction.

The grain direction is encoded in the `fuzz` field: `fuzz` less than 0.33 corresponds to Y-direction vertical grain, used for columns and other vertical members; less than 0.66 corresponds to X-direction horizontal grain, used for beams and other horizontal members; the remainder corresponds to Z-direction horizontal grain. The WGSL implementation of direction selection and roughness mapping is:

```
    if mt == 6u {
      var p_grain: vec3<f32>;
      if mat.fuzz < 0.33 {
```

```
      p_grain = vec3(p.x, p.y * 0.12, p.z);            // Y-direction
    } else if mat.fuzz < 0.66 {
      p_grain = vec3(p.x * 0.12, p.y, p.z);            // X-direction
    } else {
      p_grain = vec3(p.x, p.y, p.z * 0.12);            // Z-direction
    }
    let wv = wood_field(p_grain);
    use_n = bump_normal(p, hit.n, wv, 0.008, 0.03);     // Subtle
       normal perturbation
    use_fuzz = 0.70;
  }
```

`wood_field` itself is an isotropic three-layer FBM:

```
 fn wood_field(p: vec3<f32>) -> f32 {
    let base  = fbm(p * 2.5, 3u);                      // Coarse
       texture
    let grain = fbm(p * 8.0 + vec3(1.7, 2.3, 3.1), 3u); // Medium
       texture
    let micro = fbm(p * 22.0 + vec3(5.1, 1.3, 4.7), 2u); // Micro-pores
    return base * 0.35 + grain * 0.40 + micro * 0.25;
  }
```

The bump perturbation parameters $\varepsilon = 0.008, s = 0.03$ are extremely subtle, causing highlights to deflect slightly at texture grooves rather than producing three-dimensional relief—simulating the smooth surface of planed, sanded, and varnished wood. Four percent of specular reflection is blended into the scattering direction to simulate the faint gloss of a varnished surface; when used together with ClearCoat, ClearCoat handles the high-level coating Fresnel highlight, while the wood's own roughness distribution and specular blend handle the shallow fiber-texture reflection.

Figure 13: Visual effect of a procedural wood material.

5.2.4 Petals

The petals in the scene must simultaneously exhibit four optical behaviors: translucency of thin sheets causing glow under backlighting; subsurface scattering imparting a soft volumetric feel; the backlit halo of surface fuzz producing edge soft-light; and inter-petal gaps allowing light to pass through unimpeded. The first three are elaborated in §4; this section discusses the implementation of alpha clipping and multi-layer superposition.

Alpha Clipping. The petal model is a triangle mesh, with each triangle mapped to the petal texture via UV coordinates. The petal edge regions have zero or near-zero values in the texture's alpha channel—fully transparent. After triangle intersection and before material evaluation, the texture alpha channel is sampled and compared against a threshold:

```
// In hit_world:
if sample_texture_alpha(mat.tex_id, uv) < 0.5 { continue; }  //
    Transparent pixel, skip
```

Alpha clipping occurs after intersection is complete but before the hit record is established, so fully transparent triangles are not treated as hits—the ray continues through them, searching for the next triangle in the same leaf node or deeper nodes in the BVH. This differs from transparent materials that alter the ray direction through refraction: alpha clipping does not change the ray's direction; it merely determines whether that triangle participates in the hit competition.

Multi-Layer Superposition. The complete scattering path of a petal is de-

termined by a three-layer structure selected by probability. The first layer is the surface fuzz layer, triggered with 15% probability; a high-roughness $\alpha = 0.65$ specular-reflection perturbation produces a soft luminous contour at the petal edge when viewed under backlighting. The second layer is the Fresnel entry, where the remaining 85% of rays enter the reflection path with probability $R(\theta)$; the waxy surface with IOR 1.4 performs fuzzy reflection at medium roughness $\alpha = 0.2$. The third layer is the thin-slab random walk SSS, where rays not triggered by the first two layers enter the medium interior with probability $1 - R(\theta)$, with parameters $\sigma_a = (0.001, 0.008, 0.003)$, $\sigma_s = 6.0$, $g = 0.7$, and a maximum of 20 steps.

In the implementation, a local normal is first constructed at the petal hit point, modulated by UV coordinates and random hashing—tangent-space pillow deformation makes the petal surface slightly convex, and per-petal random tilting avoids repetitive appearance—before entering the three-level probability branching:

```
if mt == 7u {
  // Surface fuzz layer: 15% probability, rough reflection
  if rand(seed) < 0.15 {
    let refl = reflect(normalize(rd), hit.n);
    let sheen_dir = normalize(refl + 0.65 * rand_unit_vec(seed));
    if dot(sheen_dir, hit.n) > 0.001 {
      rd = sheen_dir; ro = p + hit.n * 0.002; continue;
    }
  }

  // 3D petal geometry: tangent-space pillow deformation + per-petal
     random tilt
  let uv_center = vec2<f32>(0.5);
  let uv_dist = length(hit.uv - uv_center);
  let alpha_edge = 1.0 - smoothstep(0.0, 0.45, uv_dist);
  let thickness = mix(0.003, 0.035, alpha_edge);
  let pillow_offset = (hit.uv - uv_center) * 0.20;
  let hx = hash3(p * 50.0 + vec3<f32>(0.0, 1.7, 3.1));
  let hy = hash3(p * 50.0 + vec3<f32>(5.3, 0.0, 2.7));
  let local_perturb = vec3<f32>(
  pillow_offset.x + (hx - 0.5) * 0.25,
  pillow_offset.y + (hy - 0.5) * 0.25, 1.0);
  let petal_n = normalize(onb_local(hit.n, local_perturb));

  // Fresnel entry: IOR = 1.4, Schlick approximation
  let r0 = (1.4 - 1.0) / (1.4 + 1.0); let r02 = r0 * r0;
  let fresnel = r02 + (1.0 - r02) * pow(1.0 - abs(dot(normalize(rd),
     petal_n)), 5.0);

  if rand(seed) < fresnel {
    // Reflection path: medium-roughness fuzzy reflection
```

```
      let refl = reflect(normalize(rd), petal_n);
      let rough_refl = refl + 0.2 * rand_unit_vec(seed);
      if dot(rough_refl, petal_n) <= 0.0 { break; }
      rd = normalize(rough_refl); thr *= base_color;
      ro = p + petal_n * 0.002; continue;
    }

    // Enter thin-slab random walk SSS
    // sigma_a, sigma_s, g, free-path sampling, escape check ... (see
       §4.4.2)
  }
```

Thin-Sheet Translucency. In the simplified non-SSS petal material, the scattering direction is flipped to the opposite of the normal with `fuzz` probability, simulating light penetrating a thin sheet. The transmitted color is given by exponential absorption $T = \exp(-\sigma_a \cdot d)$, with $\sigma_a = (0.2, 2.5, 4.5)$ causing blue light to be rapidly absorbed within the thin sheet; the transmitted light is warm-toned, with more red light retained. After flipping, four steps of short-range HG scattering with $g = 0.6$ are performed on the back face to simulate a small amount of scattering blur inside the thin sheet. To simulate irregular gaps between petals, a random check is applied to the hit-point coordinate: when $\text{hash3}(\mathbf{p} \times 30) > 0.55$, the hit is skipped directly and the scattering direction is resampled, visually producing the intermittent light leaks through gaps between petals.

```
  // Thin-sheet translucency: fuzz-probability direction flip + short-
     range scattering
  if mat.fuzz > 0.0 && rand(seed) < mat.fuzz {
    scatter_n = -use_n;                                  // Flip to back
        face
    let sss_dist = abs(dot(rd, use_n)) * 0.02;          // Path estimate
        inside sheet
    thr *= exp(-vec3(0.2, 2.5, 4.5) * sss_dist);        // Transmitted
       color absorption
    // Mini random walk: 4 short-range HG scattering steps
    for var ss = 0u; ss < 4u; ss++ {
      let d_free = -log(max(rand(seed), 0.001)) / 12.0;
      sss_pos += d_free * sss_dir;
      thr *= exp(-vec3(0.3, 2.5, 5.0) * d_free);
      sss_dir = sample_hg(sss_dir, 0.6, seed);
    }
    sdir = normalize(sss_dir);
  }

  // Random gap detection
  if hash3(p * 30.0 + vec3<f32>(0.3, 0.7, 0.1)) > 0.55 {
    ro = p + rd * 0.01; continue;                        // Skip this hit
```

```
}
```

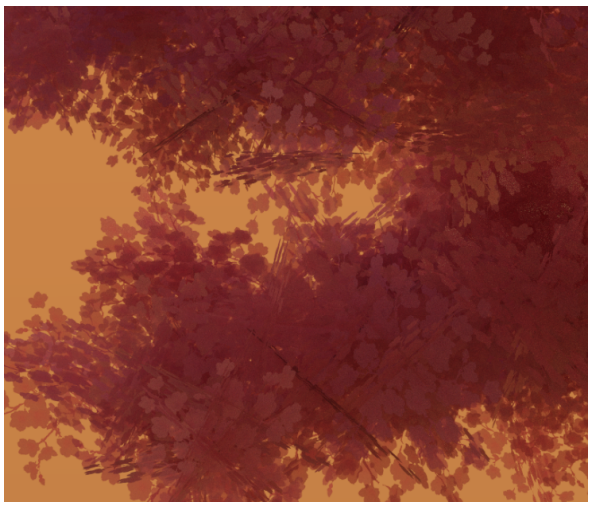

Figure 14: Visual effect of a cherry blossom cluster.

5.2.5 Fabric

Fabric materials face a seemingly contradictory visual requirement: they must simultaneously exhibit the softness of drape and the roughness of fiber weave. Softness comes from large-area low-frequency undulations that keep the reflection lobe relatively concentrated, yielding a silky luster; roughness comes from microscopic yarn crossings producing high-frequency surface texture that disrupts the integrity of specular reflection; thin-gauze or loosely woven regions should also permit subtle light transmission under backlighting.

The coexistence of softness and roughness is achieved through layered normal perturbation. The low-frequency component—from the lower FBM octave layers $\text{fbm}(\mathbf{p} \times 2.5, 2)$—simulates the folds and drape undulations of the fabric, with undulation periods on the order of tens of centimeters; the high-frequency component—from the higher FBM octave layers $\text{fbm}(\mathbf{p} \times 15, 2)$—simulates the weave texture of the yarn, with undulation periods on the order of millimeters:

```
if mt == 8u {
  let weave = fbm(p * 15.0 + vec3<f32>(0.7, 1.3, 0.3), 2u);
  use_n = bump_normal(p, hit.n, weave, 0.010, 0.02);
  use_fuzz = 0.85;
}
```

The perturbation parameters $\varepsilon = 0.010, s = 0.02$ keep the visual intensity of the bump at a perceivable but subtle level—the normal is slightly deflected by the weave texture, causing highlights to fragment at fiber crossings, while the low-frequency reflection lobe shape is preserved. A high roughness `fuzz` of 0.85 ensures that scattering

directions are always distributed within a wide cone, eliminating any recognizable specular reflection component, so that the material exhibits the softness of a purely diffuse appearance while retaining subtle surface directionality from the fabric weave. Backlit translucency is achieved through the thin-sheet translucency scheme, with `fuzz` values set in the low-to-medium range of 0.15–0.25, allowing only thin-gauze regions to subtly transmit ambient light.

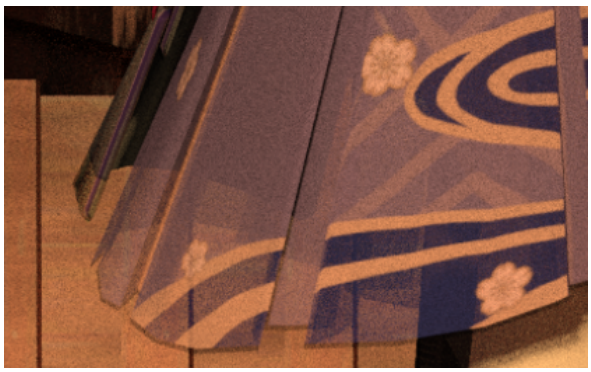

Figure 15: Visual effect of a silk-like fabric material.

5.2.6 Water Specialization

The visual core of water lies in two depth-dependent effects: the deeper the water, the darker and more blue-green its color, because red light is preferentially absorbed; and the continuous perturbation of reflection and refraction directions by surface ripples. The physical discussion of these effects appears in §4.3 (Beer absorption) and §4.2.1 (wave normal perturbation); the discussion here focuses on their parameterization within the material system.

Water detection is performed through the refractive index range check: $1.3 <$ ref_idx < 1.34, corresponding to IOR 1.333. Upon hitting the water surface, `water_wave_normal` is first called to perturb the geometric normal with three layers of sinusoidal waves. The perturbed normal then enters the Schlick Fresnel computation with $R_0 \approx 0.02$ to determine whether the ray is reflected or refracted. The reflection and refraction directions thus carry continuous variations synchronized with the wave motion—tiny changes in viewing direction or light-source direction cause different deflections of the reflected and refracted directions, producing the dynamic sparkling effect on the water surface.

After entering the water along a refraction path, Beer absorption accumulates along the light path. $\sigma_a = (0.35, 0.06, 0.025)\,\mathrm{m}^{-1}$ causes each ray passing through water to receive corresponding per-wavelength attenuation based on its path length within the water. At shallow depths of centimeters to decimeters, attenuation is nearly negligible—the water remains transparent; at medium depths of meters, red light begins to decay—the water takes on a blue-green hue; at deep depths beyond

ten meters, only blue light remains. When a ray, after multiple bounces inside the water, ultimately fails to hit the bottom and escapes, it is assigned the deep-water color $(0.02, 0.06, 0.10)$ of dark blue-green, simulating the hue of deep-water regions where light is completely absorbed.

```
if is_water {                                          // 1.3 <
    ref_idx < 1.34
  use_n = water_wave_normal(p, hit.n);                // Wave normal
     perturbation
}
// ... Schlick Fresnel decides reflection/refraction ...
if is_water {
  if (!did_reflect && hit.ff) || (did_reflect && !hit.ff) {
    in_water = true;                                  // Enter water
  }
}
// Accumulate per-bounce in subsequent path loop:
if in_water {
  thr *= exp(-vec3(0.35, 0.06, 0.025) * min(hit.t, 20.0));
}
```

Figure 16: Visual effect of a water surface with ripples and depth-dependent coloration.

5.3 A Unified View of the Fresnel Refraction Framework

All materials involving reflection/refraction bifurcation follow the same pattern: compute Schlick $R(\theta)$, with probability $U(0, 1) < R(\theta)$ taking the reflection path, otherwise taking the transmission path. The variants of the reflection path include the specular reflection of metals, the reflection of dielectrics, the highlight of clear coat, and the surface fuzz or rough Fresnel reflection of SSS petals; the variants of the transmission path include the Snell refraction of dielectrics, the dark diffuse reflection of clear coat, and the thin-slab random walk of SSS. Different materials are simply different combinations of the Fresnel reflectance with different scattering models.

6 Lighting Setup

6.1 Next Event Estimation

In path tracing, if one relies entirely on random scattering to chance upon a light source, the overwhelming majority of paths will terminate via Russian roulette before reaching any light source. Since light sources typically subtend extremely small solid angles in the scene, the probability of a randomly sampled direction from an arbitrary hit point striking a light source is nearly zero, causing the image to be drowned in high-variance noise. Next event estimation (NEE), at each surface scattering event, explicitly selects a light source and shoots a shadow ray to test visibility, incorporating the light source's direct contribution at that point into the Monte Carlo estimate. This operation is an additional deterministic step inserted into the standard path tracing loop—it does not rely on the luck of random scattering directions, but actively incorporates the light-source direction into the sampling.

The Monte Carlo estimate of NEE requires computing the PDF of sampling the direction toward the light source from the hit point. For a spherical light source, the algorithm is: uniformly randomly select one light sphere from among all light sources; uniformly sample a point on the surface of that sphere as the light-source sample point; construct the vector from the hit point toward that point as the shadow-ray direction. The solid-angle PDF of observing that light-source direction from the hit point is:

$$p(\omega) = \frac{\|\mathbf{p}_l - \mathbf{p}\|^2}{\cos\theta_e \cdot 4\pi R^2} \cdot \frac{1}{N_{\text{lights}}} \tag{30}$$

where R is the radius of the light-source sphere, θ_e is the angle between the light-source surface normal and the outgoing direction, and N_{lights} is the total number of light sources. The factor $1/N_{\text{lights}}$ in the denominator originates from the uniformly random selection among all light sources.

The shadow-ray emission direction is $\omega_i = (\mathbf{p}_l - \mathbf{p})/\|\mathbf{p}_l - \mathbf{p}\|$. A scene intersection test is launched along this direction with $t_{\max} = \|\mathbf{p}_l - \mathbf{p}\| - \varepsilon$ as the maximum distance. If any other object is hit before reaching the light source, the light source is occluded, and the NEE contribution is zero. If not occluded, the light source's contribution at the hit point is fused with the BRDF sampling result of the scattering direction through the MIS weight of §1.4.

```
    // NEE: Select a light source for explicit sampling
    if light_count > 0u {
      let li = u32(rand(seed) * f32(light_count)) % light_count;
      let ls = spheres[lights[li]];
      let light_pt = ls.center.xyz + rand_unit_vec(seed) * ls.radius;
      let to_light = light_pt - p;
      let dist = length(to_light);
```

```
      let wi = to_light / dist;

      // Shadow ray: test occlusion
      let lhit = hit_world(p + n * 0.001, wi, 0.001, dist - 0.001, r_time
          );
      if lhit.t >= dist - 0.002 {                          // Not occluded
        let cos_emitter = max(abs(dot(rand_dir, -wi)), 0.001);
        let area = 4.0 * PI * ls.radius * ls.radius;
        let light_pdf = (dist * dist / (cos_emitter * area)) / f32(
            light_count);
        let bsdf_pdf = max(cos_recv, 0.001) / PI;
        let mis_w = light_pdf / (light_pdf + bsdf_pdf);  // Balance
            heuristic weight
        col += thr * surface_color * light_color * cos_recv / (PI *
            light_pdf) * mis_w;
      }
    }
```

6.2 Sun Light Source Configuration

The main light source of the scene is a large-radius spherical light source, placed hundreds of units away from the scene center, simulating the low-angle illumination of the setting sun. The material type of the spherical light source is emissive, and its albedo is its radiance. Unlike a point light or directional light, a spherical light source has a nonzero angular diameter—when sampling points on the light source surface from a hit point in NEE, the sample points are distributed over the spherical surface rather than concentrated in a single direction, so the resulting shadows have soft penumbral transitions rather than ideal hard shadows.

The radiance of the sun light source is set to extremely high values—typically all three channels are on the order of 10^3 to 10^4—far exceeding the physical solar constant, because the Monte Carlo integration of path tracing at low SPP requires high-brightness light sources to keep the direct illumination contribution from being drowned in noise. The radius of the light-source sphere is sufficiently large (approximately 0.3 times the scene size) so that the solid angle observed from any hit point in the scene is not excessively small—too small a solid angle leads to highly unstable PDF estimates in NEE.

The sun direction $\mathbf{s}$ is a unit vector, set to $(0.692, 0.207, -0.692)$ in the scene render—the sun at dusk is low in the western sky, at an elevation angle of approximately 12°. The sun direction is used in two places in the shader. The first is in the shadow ray of NEE: if the shadow ray is unoccluded and the angle between its direction and the sun direction is less than approximately 18° ($\cos > 0.95$), the emitted color is multiplied by the golden tint $(1.0, 0.65, 0.22)$—low-angle sunlight passes

through a thicker atmospheric layer, where short-wavelength blue light is filtered out by Rayleigh scattering, and the direct light reaching the ground is a warm orange. The second use is in sky rendering, where the sun direction determines the position of the glow and the solar disk on the screen.

```
// Warm tint from sun direction in NEE
if dot(wi, normalize(sun_dir.xyz)) > 0.95 {
  light_color = light_color * vec3<f32>(1.0, 0.65, 0.22);
}
```

6.3 Ambient Light and Sky Rendering

NEE only provides direct illumination from explicit light sources; rays that miss all objects—escaping to infinity or passing through all scene objects—land on the sky background. The radiance of the sky serves as the termination value for the miss branch in the path tracing loop, equivalent to the ambient lighting of the scene. Moreover, sky radiance reflected by surfaces contributes to indirect illumination, refracted through water bodies contributes to underwater scattered light, and attenuated by atmospheric fog contributes to distant haze color.

The radiance distribution of the sky is a function of the viewing direction: the zenith direction is deep blue, the vicinity of the horizon is orange-gold, and the below-ground direction is dark brown. This color stratification simulates the physical characteristics of atmospheric scattering at dusk—at the zenith, the short light path is dominated by Rayleigh scattering (with λ^{-4} wavelength dependence), appearing deep blue; near the horizon, the long light path nearly completely scatters away short wavelengths, leaving only the warm components of Mie scattering and aerosol scattering. The sky radiance uses a three-level gradient with smoothstep piecewise transitions:

```
fn sky_color(rd: vec3<f32>) -> vec3<f32> {
  let y = rd.y;
  let zenith  = vec3<f32>(0.015, 0.025, 0.12);      // Zenith: deep
     blue
  let mid_sky = vec3<f32>(0.55, 0.12, 0.28);         // Mid-sky: warm
      purple
  let horizon = vec3<f32>(0.98, 0.42, 0.12);         // Horizon:
     orange-gold
  let r_dark  = vec3<f32>(0.12, 0.04, 0.03);         // Below-ground:
      dark brown

  var sky_base: vec3<f32>;
  if y > -0.15 {
    let t = (y + 0.15) / 1.15;
    if t < 0.38 {
      sky_base = mix(horizon, mid_sky, smoothstep(0.0, 0.38, t));
```

```
        } else {
          sky_base = mix(mid_sky, zenith, smoothstep(0.38, 1.0, t));
        }
      } else {
        let t_down = smoothstep(0.0, 1.0, clamp((-y - 0.15) * 2.5, 0.0,
            1.0));
        sky_base = mix(horizon, r_dark, t_down);
      }

      // Solar glow and disk
      let sun_cos = max(dot(rd, sun_dir), 0.0);
      let glow_wide = pow(sun_cos, 6.0) * vec3<f32>(1.0, 0.3, 0.05) *
          2.0;
      let sun_disk  = pow(sun_cos, 400.0) * vec3<f32>(1.0, 0.85, 0.4) *
          15.0;
      return sky_base * 0.25 + glow_wide * 0.35 + sun_disk;
    }
```

The visual appearance of the sun in the sky consists of two components. The wide glow uses a power of $\cos^6\theta$ and is visible over approximately 30° around the sun—simulating the forward-scattering enhancement effect of Mie scattering in the atmosphere: the sky close to the sun direction is brighter and warmer than the sky farther from the sun direction. The solar disk uses an extremely high power of $\cos^{400}\theta$ and is effective only within an extremely narrow cone of approximately 2° around the sun direction itself—simulating the direct radiation of the solar disk; the multiplier of 15.0 makes it significantly exceed the sky base color in HDR rendering, and after Reinhard tone mapping it appears as the bright core of the light source.

7 Final Scene: Narukami Shrine

7.1 Rendering Parameters

The final render is centered on the publicly distributed 3D model of the Narukami Shrine from Genshin Impact, paired with the Kamisato Ayaka character model. The scene consists of 40 texture files, 8 material types, and a single spherical sun light source. The rendering parameters are as follows:

Table 1: Final Scene Rendering Parameters

Parameter	Value
Resolution	7680×4320 (8K)
Samples per pixel (SPP)	256
Maximum bounce depth	50
Tile size	512×512
Samples per batch	8
Sun direction	$(0.692, 0.207, -0.692)$
Light source type	Single spherical light, radiance $(2000, 1400, 600)$
BVH construction method	16-bin SAH, leaf-node cap 8 primitives
Scene triangle count	$\sim$1,500,000
Scene sphere count	4 (1 light + 3 auxiliary)
Material type count	8
Texture count	40

Table 2: Key Material Physical Parameters

Material	Parameter	Value
Water (Material 33)	IOR, σ_a	1.333, $(0.35, 0.06, 0.025)$
Glass (Material 17)	IOR, fuzz	1.45, 0.0
Metal (Material 8)	fuzz	0.08
Metal (Material 16)	fuzz	0.2
Metal (Material 35)	fuzz	0.3
ClearCoat (Material 4 etc.)	R_0, fuzz	0.06, 0.20–0.25
Stone (Materials 5/10/15/21–30/37/38)	ε, s	0.010, 0.04
Wood (Materials 1/2/3/7/11/13/18/34)	ε, s	0.008, 0.03
Linen (Material 4)	ε, s	0.010, 0.02
Petal (Materials 14/31)	σ_s, g, IOR	6.0, 0.7, 1.4

7.2 Final Renderings

You can find them somewhere else.

Was it one's thoughts that drew him to my dreams?

Had I known it a dream, one would not have awakened.

The End.

References

[1] J. T. Kajiya. *The Rendering Equation.* ACM SIGGRAPH Computer Graphics, 20(4):143–150, 1986.

[2] P. Shirley, T. D. Black, S. Hollasch. *Ray Tracing in One Weekend* (v4.0.1). 2024. `https://raytracing.github.io/`

[3] P. Shirley, T. D. Black, S. Hollasch. *Ray Tracing: The Next Week* (v4.0.0). 2024. `https://raytracing.github.io/`

[4] P. Shirley, T. D. Black, S. Hollasch. *Ray Tracing: The Rest Of Your Life* (v3.2.3). 2020. `https://raytracing.github.io/`

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

[6] E. Veach, L. J. Guibas. *Optimally Combining Sampling Techniques for Monte Carlo Rendering.* ACM SIGGRAPH, 1995.

[7] C. Schlick. *An Inexpensive BRDF Model for Physically-Based Rendering.* Computer Graphics Forum, 13(3):233–246, 1994.

[8] T. Möller, B. Trumbore. *Fast, Minimum Storage Ray-Triangle Intersection.* Journal of Graphics Tools, 2(1):21–28, 1997.

[9] D. J. MacDonald, K. S. Booth. *Heuristics for Ray Tracing Using Space Subdivision.* The Visual Computer, 6(3):153–166, 1990.

[10] L. G. Henyey, J. L. Greenstein. *Diffuse Radiation in the Galaxy.* The Astrophysical Journal, 93:70–83, 1941.

[11] E. Reinhard, M. Stark, P. Shirley, J. Ferwerda. *Photographic Tone Reproduction for Digital Images.* ACM Transactions on Graphics, 21(3):267–276, 2002.