

Real-Time Raytracer for Translucent Materials

by

Marc Gluyas

A thesis
submitted to the Victoria University of Wellington
in partial fulfilment of the
requirements for the degree of
Master of Science
in Computer Graphics.

Victoria University of Wellington
2025

Abstract

For this work, I have developed an interactive path tracer with first-class support for physically-based rendering of translucent materials. The software implements a fully GPU-driven raytracing pipeline with DirectX Raytracing, synthesising several compute-based algorithms to render non-local subsurface scattering effects based on a blue noise surface sampling model[2, 8, 9]. The pipeline is easily extendable with additional material models and exposes a clean API, hiding complex device interfacing and resource management while mirroring low-level hardware raytracing features. In this thesis, I will first review the relevant theoretical background and provide an overview of DirectX 12 and DirectX Raytracing. I then provide a detailed description of the raytracing system and its implementation details. Finally, I share my results demonstrating the wide range of material properties and appearances that are supported by the software and give ideas for its future development.

Acknowledgments

My parents: Your unconditional support

My supervisor: Your encouragement and patience

My friends: You inspire

Contents

1	Introduction	1
2	Theoretical Background	3
2.1	Physically-Based Rendering	3
2.1.1	The Rendering Equation and BRDF	4
2.1.2	Numerical Solutions	5
2.2	Rendering of Translucent Materials	8
2.2.1	Volume Rendering	9
2.2.2	Analytical and Stochastic BSSRDF Methods	11
2.3	Blue Noise Surface Sampling	13
3	DirectX 12	17
3.1	Direct3D 12	17
3.1.1	Shaders	17
3.1.2	Resource Management	18
3.1.3	Resource Binding	19
3.1.4	Work Submission	22
3.2	DirectX Raytracing	23
3.2.1	Acceleration Structures	24
3.2.2	Raytracing Shaders	25
3.2.3	Shader Execution Model	29

4	Implementation	33
4.1	System Overview	33
4.2	Software Architecture	34
4.2.1	Modules	35
4.2.2	Utilities	35
4.2.3	Execution Model	36
4.3	Geometry Processing	37
4.3.1	Mesh Loading	37
4.3.2	BLAS Construction	38
4.3.3	TLAS Construction	39
4.4	Raytracing Procedure	40
4.4.1	Path Tracing	40
4.4.2	Hit Shaders	42
4.5	Blue Noise Sample Point Generation	43
4.5.1	Mesh Preprocessing and Instancing	44
4.5.2	Initial Random Point Generation	46
4.5.3	Hashtable Construction	46
4.5.4	Sample Point Selection	47
4.6	Diffuse Reflectance Evaluation	48
4.6.1	Irradiance Sampling	48
4.6.2	BSSRDF Integration	49
5	Results	51
6	Conclusion and Future Work	63
6.1	Potential Expansions	64
6.1.1	Software Features	64
6.1.2	Physically-Based BRDF Models	64
6.1.3	Accelerated Irradiance Sample Integration	64
6.1.4	Biophysical BSSRDF	65
6.1.5	Spectral Rendering	66

Chapter 1

Introduction

Accurately rendering translucent materials is a computationally challenging problem that is important in a wide range of industries including entertainment, computer-aided design, and biomedical imaging. While approximate and simplifying solutions have been developed[3], faithful rendering of translucent materials is typically left to offline, CPU-bound path tracers that can simulate global illumination[7, 12]. The advent of hardware-accelerated raytracing, and its continued development to date, promises to buck this trend with a new generation of photorealistic renderers, leveraging the new hardware and APIs to push physically-based rendering closer to real-time performance.

DirectX Raytracing was announced in early 2018, heralding the launch of Nvidia's RTX series and their Turing architecture later the same year. Turing introduced additional specialised processing units to the dye, including RTX cores for raytracing and tensor cores for AI workloads, which have since become a standard inclusion on Nvidia's high-end GPUs. In 2020 AMD followed suit with the inclusion of specialised raytracing cores with the RDNA 2 architecture and subsequent architectures. At the same time, Vulkan standardised its own raytracing extension, and in 2022 Apple did the same with their Metal API. RDNA 2 would also become the chosen architecture for the most recent Playstation and Xbox consoles, unifying

the technology across all vendors and platforms.

While support has grown across the board, adoption of the technology remains relatively narrow. Usage of raytracing in modern video games is typically only supplementary to rasterisation, while Nvidia continues to market the technology with RTX mods and re-releases for retro games: *Quake II RTX* (2019), *Portal RTX* (2022), the upcoming *Half-Life 2 RTX* utilising their AI-powered *RTX Remix* modding toolkit, and of course, *Minecraft RTX* (2020).

While these demos have become increasingly impressive, there has been relatively little application of the technology for GPU-based renderers. Mitsuba 3[7] is able to leverage the hardware through its OptiX backend, a CUDA-based raytracing framework that received support for RTX. Another Nvidia demo *Marbles at Night* (2020) is still perhaps the most impressive use of the hardware for path tracing; a physics-based game with real-time global illumination and physically-based materials.

While currently there has not been a true “killer-app” for hardware raytracing, it is clear that the technology is here to stay. As consumer adoption slowly increases and technical advances are made, it is hard to imagine a future where real-time raytracing does not become a staple of the industry. In this work, I explore how this technology can be used at the lowest level to build an interactive, extendable raytracing system, and how it can be leveraged to advance the rendering of translucent materials.

Chapter 2

Theoretical Background

Physically-based rendering (PBR) is an approach to computer graphics that began to develop in the 1980s. In essence, physically-based approaches seek to faithfully recreate the visual world by studying and applying the physical laws of light. The earliest steps into PBR were made over several years with foundational research by Whitted[13], Cook, Torrance[4], and Kajiya[10]. From this work and that of many others has emerged a formalization of the language and mathematics of rendering, which now serves as the backbone for physically based rendering.

In the first section, this chapter will first review the bread and butter of PBR, before diving into the theory and problems of volumetric rendering and subsurface scattering in the second.

2.1 Physically-Based Rendering

Rendering is fundamentally about understanding how light interacts with the world around us - we only see things as they affect light travelling into our vision. The huge variety of visual phenomena reveals its complexity: Pigments take on the colours of light they reflect; electric currents cause gasses in tubes to emit their own. Mirrors show us the world behind us and glasses shape it to fit our retinas. Our faces can flush from embar-

rassment or go pale with shock. Metals lustre and silks shine. Water in a glass projects its ripples onto the ceiling above and table below. Tiny raindrops spread sunlight into a rainbow; the mighty pāua fracture it with their shells.

There is an ever-present, endlessly complex interplay between light and the world around us, but a single optical phenomenon colours and shapes our perception more than any other: reflection. When light interacts with opaque objects, it is either absorbed and converted into heat, or reflected, temporarily transferring its energy into electrons before being re-emitted. The colour of the object is determined by what wavelengths of light the material reflects, while the visual texture of the object is determined by the directional distribution of these reflections. Materials with rough, matte surfaces scatter incoming light randomly in all directions, while smooth, shiny materials reflect light specifically at the angle it was incident in. Most materials in the real world exhibit varying degrees of both of these properties; modelling and integrating this scattering behaviour is the basis of physically-based rendering.

2.1.1 The Rendering Equation and BRDF

The rendering equation[10] provides a general description of how light is reflected by opaque objects towards the viewer. It defines a function for L_o , the light reaching the observer along unit vector ω_o , given a point on the surface p .

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_{H^2(n)} f(p, \omega_o, \omega_i) L_i(p, \omega_i) |\omega_i \cdot n| d\omega_i \quad (2.1)$$

By evaluating the rendering equation for all points facing the viewer, we can render an image of an entire scene; hence the name.

Light is measured with a quantity called **radiance**, denoted by the subscripted L symbols, defined as the radiative power flowing along a straight line in free space. The first term in the definition, L_e , denotes ra-

diance emitted by the surface from point p towards the observer along ω_o . Few materials are emissive in general, so with the exception of lights, this term will be zero in all directions for most objects in a scene.

The integral term in the definition calculates how light is scattered towards the observer. The integral takes incident radiance (L_i) from all directions (ω_i), on the hemisphere ($H^2(n)$), then sums their contributions via scattering towards the observer (ω_o). The differential contribution of L_i scattered towards ω_o is given by its product with a cosine term, $|\omega_i \cdot n|$, and a distribution function, $f(p, \omega_o, \omega_i)$.

The cosine of ω_i with the surface normal (n) gives its differential projected area on the surface. The product of this area and L_i yields a differential **irradiance** quantity, denoted dE , defined as the radiative power density on a surface.

The function f is known as the **bidirectional reflectance distribution function** (BRDF) and completely describes the reflective and scattering properties of opaque materials. The BRDF is parametrized under p , ω_o , and ω_i , and relates the differential irradiance incoming from ω_i to differential radiance outgoing along ω_o . In other words, it gives the directional distribution of $\frac{dL_o}{dE_i}$ over each point on the surface of a material.

2.1.2 Numerical Solutions

The complexity of the rendering equation makes it impossible to solve analytically for anything but the most contrived cases. Reality is comprised of detailed objects with complex geometry and reflection functions, so robust numerical methods must be employed to even begin to attempt photorealistic rendering.

Whitted Raytracing

The raytracing algorithm is the basis for the solution to almost all rendering problems. First proposed by Whitted[13], the algorithm traces *back-*

wards along the path taken by light to reach the observer. To render an image, we need to calculate the outgoing radiance from all camera-facing surfaces arriving at the camera. In general, most of the light in the scene is scattered outside of the observer's field of view; The key idea of raytracing was to compute only the light that is relevant to the image by starting from the observer's position and tracing backwards along the light's path, rather than forwards out of the scene.

For each pixel (or sample) in the image, calculate the direction of light reaching it through the camera lens. The **primary ray** is defined as the ray originating at the position of the camera and pointing in the opposite direction to this. Using the model of a pinhole camera, each sample in the image corresponds to a unique primary ray, whose direction is substituted as $-\omega_o$ into the rendering equation. The algorithm then steps outwards along the primary rays into the scene, finding the nearest intersecting surfaces. The rendering equation is then evaluated at these points of intersection, given as p , with L_o determining the resulting colour of the samples.

Whitted's initial implementation of raytracing used the much older Blinn-Phong illumination model[1], and was only capable of considering direct illumination from a fixed number of light sources with simple, isotropic BRDFs. However, it was remarkable at the time for its ability to accurately render refractions and mirror reflections by recursively tracing secondary rays along these altered light paths. This style of renderer is now known as a Whitted raytracer.

Path Tracing

While Whitted's original algorithm only considered a discrete set of light sources, it naturally leads to a recursive algorithm that accounts for illumination from *all* objects in a scene. Despite most light in a typical scene originating from a small set of emissive sources, a significant amount can be transmitted between opaque surfaces. This can be expressed as a recursive relation between L_i and L_o in the rendering equation; for all points p

and p' that are facing each other:

$$L_i(p, \omega) = L_o(p', -\omega), \text{ where } p' = p + t\omega, \quad t > 0 \quad (2.2)$$

This **indirect illumination** leads to a recursive process known as **diffuse interreflection** where light scattered by one surface illuminates others. This is apparent in the rendering equation; because ω_i is integrated over the entire hemisphere, any set of surfaces that are facing each other will have L_i terms that depend on L_o terms of the others.

This process accounts for a large proportion of visible lighting effects. It is the main reason why a single lamp can light an entire room and why dark corners appear to expand gradually, rather than exist behind an abrupt boundary. Deciding what to paint the walls really does matter, because they colour the lighting conditions of everything they surround.

Path tracing, first introduced by Kajiya[10], implements diffuse inter-reflection by modifying the raytracing algorithm to sample incident radiance from indirect light paths. Rather than integrating L_i from direct sources only, it evaluates the rendering equation by randomly sampling ω_i over the hemisphere. These discrete samples are then used to generate **secondary rays**, with origin p and direction ω_i , which recursively invoke the raytracing procedure to calculate further samples of L_i . This gives a Monte-Carlo solution to the rendering equation:

$$L_o(p, \omega_o) \approx L_e(p, \omega_o) + \frac{1}{N} \sum_{j=1}^N f(p, \omega_o, \omega_i^j) L_i(p, \omega_i^j) |\omega_i^j \cdot n|, \quad \omega_i \sim H^2(n) \quad (2.3)$$

Secondary rays are recursively sampled until they hit an emissive light source or reach a fixed depth limit. Each level of indirection has diminishing contributions to the final result, so the depth limit is generally kept low due its exponential effect on computational complexity. For a single light path with D bounces, the differential contribution to L_o is given by the product of BRDF and cosine terms and the L_e term of the emissive

source:

$$d^D L_o(p^1, -\omega^1) = \frac{1}{N^D} \left(\prod_{j=1}^D f(p^j, -\omega^j, \omega^{j+1}) |\omega^{j+1} \cdot n^j| \right) L_e(p^{D+1}, -\omega^{D+1}) \quad (2.4)$$

Path tracing remains the gold standard for rendering. It is generally considered the most faithful solution to the rendering equation as it has unbiased convergence and no theoretical upper bound on accuracy. However, its high computational complexity and low entropy per sample make it impractical for most applications and is seldom used without heuristic modifications.

2.2 Rendering of Translucent Materials

So far we have discussed rendering opaque materials using the rendering equation with the BRDF. The BRDF model assumes that light is only reflected, absorbed, or transmitted locally, at surface boundaries, and flows freely through space otherwise. However, participating media such as smoke and translucent materials such as marble or skin break this assumption. These materials partially transmit radiance while volumetrically scattering and absorbing it. The BRDF is unsuitable because it cannot account for the non-local effects of volumetric scattering, where radiance can be scattered in arbitrary directions at all points along the ray. Volumetric scattering contributes significantly to the appearance of translucent materials, so other methods must be used to render them accurately.

While volumetric scattering is functionally the same in both participating media and translucent materials, they warrant different approaches to their implementation. Specifically, **subsurface scattering** refers to volumetric scattering below the surface of a translucent material, and is the primary focus of this research; we will not consider participating media. We will also not consider emissive volumetric materials such as plasma.

2.2.1 Volume Rendering

There are two major volumetric effects that need to be considered for translucent materials: **absorption** and **scattering**.

Absorption refers to radiation that is absorbed by particles in the medium and converted into another form of energy such as heat. This reduces the radiant energy transmitted through the medium, visually darkening it and casting shadows when thick enough. Absorption in a material is quantified by the absorption cross-section, $\sigma_a(p, \omega)$, a function of position and direction. It gives the proportion of radiance absorbed per unit distance travelled along ω through p .

Scattering refers to radiation that collides with particles in the medium and is re-emitted in another direction. Similar to absorption, scattering is quantified by the scattering cross-section, $\sigma_s(p, \omega)$, the proportion of radiance scattered per unit distance travelled along ω through p .

The direction that light is scattered in is probabilistic and is given by the material's **phase function**, $p(p, \omega_i, \omega_o)$, which defines a normalized probability density function for each pair of incoming and outgoing light directions. For many materials the phase function only depends on the angle between the two directions – such materials are considered **isotropic** as the scattering distribution is rotationally symmetric about the incident ray. Scattering leads to two distinct visual effects: **out-scattering** and **in-scattering**.

Out-scattering refers to attenuation in radiance due to being scattered in other directions. Because absorption and scattering both exponentially attenuate radiance along a ray, they are combined into a single term called the extinction coefficient, $\sigma_t = \sigma_a + \sigma_s$. This gives a single expression for the rate of attenuation in radiance along differential length dt :

$$dL_o(p, \omega_o) = -\sigma_t(p, \omega_o)L_i(p, -\omega_o)dt$$

Integrating this quantity over the length of a ray and raising it gives the total attenuation for a finite distance, T_r , the **beam transmittance**.

$$L_o(p', \omega_o) = T_r(p \rightarrow p') L_i(p, -\omega_o)$$

where

$$T_r(p \rightarrow p') = e^{-\int_0^d \sigma_t(p+t\omega, \omega) dt}, \quad p' = p + d\omega$$

$T_r(p \rightarrow p + d\omega)$ can be thought of as a probability distribution of the distance d travelled by a photon before being absorbed or scattered by the medium. For a uniform medium, the mean of this distribution $1/\sigma_t$ gives the material's **mean free path**, which is a useful metric for determining the overall scale of volumetric effects. Another useful metric, the **albedo**, $\rho = \sigma_s/\sigma_t$, gives the probability of scattering vs absorption. Light travelling through a high albedo medium undergoes more scattering events on average which contributes to a greater degree of diffuse illumination, while low albedo media is characterised by low-degree scattering which correlates more strongly with the material's phase function.

In-scattering refers to the radiance that is added along a ray due to being scattered from other directions. Every scattering event from radiance in another direction has a probability, given by the phase function, to scatter light in the direction in question and make a positive contribution to its radiance. The differential change in outgoing radiance at p along ω_i due to in-scattering can be calculated by integrating incident radiance and the phase function over the sphere.

$$dL_o(p, \omega_o) = \sigma_s(p, \omega_o) \int_{S^2} L_i(p, \omega') p(p, \omega_i, \omega_o) d\omega_i$$

By taking the sum of these differential terms we get the **radiative transfer equation** (RTE) which describes the transfer of radiance through a volumetric medium. It is usually expressed with the advection operator,

$(\omega \cdot \nabla)L(p, \omega)$, which denotes the scalar gradient of radiance along the vector ω at each point in the volume.

$$(\omega \cdot \nabla)L(p, \omega) = -\sigma_t L(p, -\omega) + \sigma_s(p', \omega) \int_{S^2} L(p, \omega') p(p, \omega', \omega) d\omega' + s(p, \omega)$$

This is an integro-differential equation that can be solved to give the radiance distribution L by substituting known light sources as boundary conditions. The final term s , the source term, defines the distribution of volumetric emission within the material. We are not considering volumetric emission specifically, but it has been included here as it is commonly used in light transport models even for materials that are not emissive.

Finally, we can integrate the scattering and absorption effects along the ray to give an expression for the radiance transmitted to one point from another.

$$\begin{aligned} L_o(p', \omega_o) &= T_r(p \rightarrow p') L_i(p, -\omega_o) \\ &+ \int_0^t T_r(p'_t \rightarrow p') \sigma_s(p'_t, \omega_o) \int_{S^2} L_i(p'_t, \omega_i) p(p'_t, \omega_i, \omega_o) d\omega_i dt' \end{aligned}$$

where $p'_t = p + t'\omega_o$.

2.2.2 Analytical and Stochastic BSSRDF Methods

Evaluating the RTE is computationally very difficult due to the recursive nature of the scattering process. Rather than using the RTE directly for translucent objects, it is factored into an extension of the BRDF, the **bidirectional subsurface scattering and reflectance distribution function** (BSSRDF). Subsurface scattering refers to the process of light entering the surface of a translucent material at one point and being re-emitted due to scattering at another. The BSSRDF expresses the relationship between the incident and exitant light at these two points in terms of differential irradiance. It is used with a modified version of the rendering equation which integrates over the surface of the object as well as the hemisphere of incident radiance.

$$L_o(p_o, \omega_o) = L_e(p_o, \omega_o) + \int_A \int_{H^2(n)} S(p_o, \omega_o, p_i, \omega_i) L_i(p_i, \omega_i) |\omega_i \cdot n| d\omega_i dA$$

The BSSRDF essentially provides a solution to the RTE between the points on the surface of a translucent material. This is useful in a rendering context, but the RTE must still be solved to produce an expression for the BSSRDF. This can be done either analytically by making simplifying assumptions about the properties of the material to give a closed-form approximation, or through Monte Carlo simulation. Most models, including Monte Carlo, make the assumption that the BSSRDF depends only on the distance between the two points. This is an important assumption as it greatly reduces the function's dimensionality, and is generally accurate when the material properties have relatively low variance at the scale of its mean free path.

Dipole Model

The dipole model for translucent materials is an analytical BSSRDF method developed by Jensen et al. It is based on a diffusion approximation of multiple scattering: that is, repeated scattering events tend towards a uniform diffusion process. Diffusion problems are well-studied and have readily available closed-form solutions. The dipole model approximates the RTE by modelling all light as coming from two point sources - one directly above the surface and one directly below, hence dipole. The radiance distribution from these two points is described analytically using the diffusion model; this is then used to calculate the BSSRDF as a ratio of incident irradiance to exitant radiance.

While it is true that multiple scattering in most materials does tend towards an isotropic distribution, the diffusion assumption loses relevance for low-albedo media where primary, secondary, and low-order scattering are the primary contributors to the visual appearance.

Monte Carlo

Monte-Carlo BSSRDF methods[5] enable the potential for a full simulation of the behaviour of light at the photon level; their primary purpose is to give as close to ground truth results as possible at the cost of time-consuming simulation. They are best suited to materials that cannot be easily modelled by analytical methods, such as biological tissues, which contain complex microscopic substructures that interact with the light in addition to typical volumetric effects. Monte Carlo methods simulate individual light paths travelling through the medium, allowing them to account for the global subsurface scattering properties caused by microscopic structures.

2.3 Blue Noise Surface Sampling

In addition to complex BSSRDF models, evaluating the BSSRDF rendering equation requires integrating irradiance over the surface of the object. Like the BRDF rendering equation, the area integral in the BSSRDF rendering equation is generally unsolvable and requires stochastic methods to compute. This requires sampling the irradiance of random points on the surface.

Unlike sampling the hemisphere which has a simple closed-form solution for all cases, surface sampling requires taking the entire surface into consideration to give an unbiased distribution. This necessitates the pre-computation of sample points for translucent media in a path tracer.

Blue noise is a specific family of point distributions that is specifically useful for translucent materials. A set of blue noise samples is defined as a random set of points that all satisfy a minimum distance requirement; this removes higher frequencies in the signal and lowers variance in the distribution. The algorithm of drawing random points until it meets the distance constraints is known as **Poisson disc sampling**.

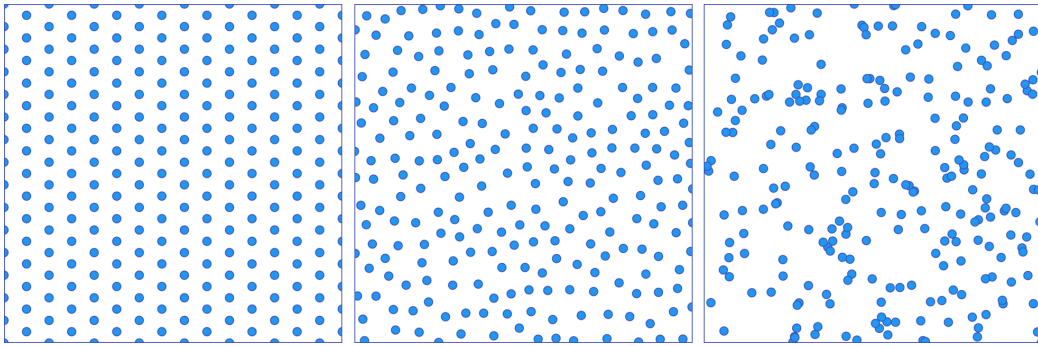


Figure 2.1: Comparison of different surface sampling algorithms of 240 points. From left to right: Offset grid; Poisson disc '*blue noise*'; Uniform random '*white noise*'. The offset grid and other latticed sampling algorithms achieve near-uniform density but are prone to directional artifacts due to their regular structure. White noise is statistically unbiased but produces highly oversampled clusters. Blue noise strikes a balance between these approaches by mitigating global structural repetition while creating a workable upper bound on local point density.

Blue noise's reliable uniformity makes it a pragmatic choice for integrating the BBSRDF; by setting radius between samples at some ratio of the mean free path, guarantees can be made about the quality of the sampling distribution everywhere on the surface. This avoids the issues with the uniform random distribution, which produces highly oversampled clusters and undersampled vacuums.

Chapter 3

DirectX 12

DirectX 12 (DX12) is a suite of low-level, high-performance multimedia APIs for Windows. Released in 2015, DX12's major addition was the Direct3D 12 graphics API, a significantly lower-level and more performance-focused API than its predecessor. Around the same time, Khronos's Vulkan and Apple's Metal APIs were released, marking an industry shift away from the high-level, state-based graphics APIs of OpenGL and Direct3D 11.

3.1 Direct3D 12

Direct3D 12 (D3D12) is a modern, high-performance graphics API similar to Khronos' Vulkan and Apple's Metal. Compared to previous APIs such as DirectX 11 and OpenGL, D3D12 exposes much lower level manipulation of GPU resources and execution and combines rasterization and general-purpose GPU programming into a single API.

3.1.1 Shaders

Shaders function similarly in D3D12 as with other graphics APIs. The rasterization pipeline has several optional and required programmable stages

which are implemented with shaders.

Unlike OpenGL, shaders in D3D12 no longer need to be compiled by the driver at runtime. DX12 includes a standalone shader compiler, DXC, which compiles HLSL to a standardized bytecode format that is supported on all DX12 platforms. This unifies the development of shaders and includes them in part of the regular application build process.

Vertex shaders and **pixel shaders** function as expected. The memory layout and format of vertex attributes is specified in the pipeline state and then passed to the vertex shader. The vertex shader outputs the final vertex position and any data required for shading. The pixel shader interpolates any data passed from the vertex shader and outputs the final pixel colours for the geometry. The rasterization pipeline is only very minimally used in our raytracer as it is separated from the raytracing pipeline.

Compute shaders are well integrated with the D3D12 rasterization pipeline as they can easily access all resources used by other shaders. Compute shaders are launched in a 1, 2, or 3-dimensional grid; all grid cells are assigned to threads on the GPU and are executed in parallel. Compute shaders are able to perform a much broader set of operations than the raster pipeline shaders and are able to manipulate resources directly on the GPU without copying to system memory. This removes a major performance bottleneck while leveraging the GPU's parallel compute power to greatly accelerate many general-purpose and certain rendering workloads that might otherwise be performed by the CPU.

3.1.2 Resource Management

D3D12 has a complex resource model to enable efficient management of device memory, paging, and bandwidth. Resources are allocated on so-called **heaps** in device-accessible memory, of which there are several types

to reflect its expected usage patterns. **Default heaps**, as the name suggests, provide the best performance characteristics for device reads and writes, but are generally not accessible by the host. **Upload heaps** and **readback heaps** are accessible by both the host and device, but are optimized for transferring data from host to device, and device to host, respectively. Custom heaps can also be created with a mixture of properties for other specific usage scenarios.

Applications will typically prefer to work with data either in a default heap (on the device) or in main system memory (on the host), and transferring data from one to the other will be implemented by first temporarily writing to an upload or readback heap, with a subsequent copy from there to its target location. This pattern is necessary for common procedures such as uploading geometry and textures and returning data from compute shaders.

D3D12 also allows the application to have fine-grained control of device virtual memory to enable efficient reuse of memory and streaming.

3.1.3 Resource Binding

D3D12 has a flexible but complex resource binding model, supporting a range of data access patterns in shaders. Shaders specify their resource requirements through objects called **root signatures**, and the resources are supplied to the shader through **descriptors**.

Descriptors

Descriptors are opaque, tagged reference types that describe resources on the GPU. In general, they hold, among other things, the type, dimensions, format, and memory location of arbitrary device resources. There are several types of descriptors, depending on the kind of resource they reference and its required access patterns. Descriptors are created and manipulated indirectly through the API, and form the basis for passing parameters to

shaders and certain graphics commands.

Shader resource views (SRVs) provide read-only access to most resource types including structured arrays (buffers), and textures.

Unordered access views (UAVs) are much the same as SRVs, but also allow the shader to perform unordered writes to the resource. Unordered writes are not synchronized between shader threads, although atomic operations do exist for specific operations, depending on platform compatibility.

In particular, SRVs and UAVs enable hardware-accelerated access operations for certain resource types, such as filtered texture fetches.

Constant buffer views (CBVs) provide read-only access to a limited set of resource types, primarily structured or untyped data arrays.

Render target views (RTVs) and **samplers** are specialized descriptors specifically for the rasterizer's framebuffer and the sampling parameters for texture resources, respectively. RTVs are not used for shader resource binding and are instead supplied directly to the command list to receive rasterization output.

Descriptors may be stored in objects called a **descriptor heaps**, which essentially function as an array of descriptors that may be accessed by shaders at runtime. The descriptor heap is used extensively for resource binding, but only a single heap may be accessed by shader programs each draw call.

Root Signatures

Every shader must be associated with a root signature that specifies the resources that are required by the shader. Requirements are specified

through a set of **root parameters**, which specify how resources are bound via descriptors to global resources in the shader.

Each global resource in the shader is assigned a unique **shader register** from a virtual register space. The root parameters then map descriptors to all of the shader's registers. The actual arguments for the root parameters (**root arguments**) are submitted onto the command list before every draw call, with each argument receiving a sequentially numbered binding slot.

There are several different types of root parameters that specify the mechanism by which the resource descriptors are mapped to their shader registers and accessed by the shader. This allows for a high degree of flexibility over data access patterns and allows for the application to trade off access speed and memory footprint for each resource used by the shader.

Descriptor handles are the most common type of root parameter and are effectively pointers into the descriptor heap. This maps the descriptor to the shader register through an indirection, which is useful for patterns that use a set of separate but identically laid out descriptor heaps to perform their operations on multiple sets of data.

Descriptor arrays are similar to descriptor handles but point to a contiguous range of descriptors in the descriptor heap. Descriptor arrays are bound to a contiguous range of shader registers in the shader. In general, descriptor arrays are preferable to descriptor handles as both take up the same space in the root signature, however, it requires additional coordination from the application to ensure that relevant descriptors are grouped and updated together in their descriptor heap.

Root descriptors map descriptors to shader registers by storing them directly in the root signature. This uses twice as much space in the root signature as a descriptor handle but does not require the descriptor to be

present in the descriptor heap, and removes a level of indirection taken by the shader to access the resource. This makes it trivial to dynamically set the descriptor every draw call as it only requires changing the root argument instead of writing to the descriptor heap. These should be used sparingly, and are most appropriate for highly trafficked resources such as geometry data.

Root constants are an additional option specifically for CBVs, which inlines the constant buffer data directly into the root signature rather than using a descriptor. This reduces the level of indirection to zero so the data is directly accessible by the shader, but can only be used for a very small amount of data.

3.1.4 Work Submission

In D3D12 graphics and compute commands are recorded on the host ahead of time via a command list, and then submitted to the device for asynchronous execution. This vastly differs from the approach taken by previous APIs, where device state was global between CPU threads and workload execution timing and synchronisation was opaquely decided by the driver.

This approach has several advantages. The ability to record multiple command lists with independent, localized device state, as well greatly increases the potential for multithreaded applications.

To support asynchronous device execution, the host is now required to explicitly synchronize with the device using fences. Fences are 64-bit semaphores that can be signalled by the device on completion of each command list. The typical usage pattern is for each command list to increment the fence's value by 1, which allows the host to see how much work has been completed, and optionally wait for work to be completed.

3.2 DirectX Raytracing

DirectX Raytracing (DXR) is a new API for DirectX 12 that was introduced alongside Nvidia RTX to support the new hardware raytracing capabilities. Raytracing has significantly different requirements to rasterization, chief among them being the necessity to have a complete description of the scene geometry to render indirect illumination. In rasterization, geometry is explicitly specified and projected directly onto the screen at render time; any geometry that is not directly visible has no effect on the final image. This has been a useful assumption that in many respects has made real-time rendering possible up until this point. Advances in hardware and shading techniques have produced increasingly accurate approximations of indirect lighting effects such as reflection and refraction, but at its core, the rasterization pipeline is primarily built around shading directly visible geometry.

Another major requirement for the new API is supporting multiple material shaders simultaneously. Each draw call in the rasterization pipeline passes all geometry through a single shader program that is loaded onto GPU execution units. This creates the assumption that all shading models required for the visible geometry are known ahead of time and that vastly different materials will require separate render passes. This is not possible in raytracing, again due to the indirect visibility requirements; all material shaders must be accessible on demand for their indirect lighting contributions to be accurately rendered.

DXR also supports raytracing against implicitly defined geometry, a quintessential raytracing feature that was only possible through costly triangulation procedures in the rasterization pipeline. These requirements have led to the addition of multiple new objects and concepts to D3D12, as well as greatly more flexible compute capabilities that may be leveraged by raytracing and traditional workloads alike.

3.2.1 Acceleration Structures

Unlike rasterization, raytracing with real-time performance requires the use of specific acceleration structures to store the scene geometry. Where geometric primitives have traditionally been fed directly into the rasterization pipeline, DXR introduces a more complex API to support the creation and manipulation of acceleration structure objects. Two new resource types, **bottom-level acceleration structures** and **top-level acceleration structures** are used to represent geometry for raytracing in an opaque, driver-defined format enabling hardware-accelerated scene traversal. Acceleration structures may be created and modified through a set of new functions and subsequently bound to the raytracing shader. Range of optional build flags are available to customize the acceleration structure construction to match the application's use cases, including prioritizing trace speed, build speed, speed of incremental updates, and memory usage.

Bottom-Level Acceleration Structures

The bottom-level acceleration structure (BLAS) contains the underlying geometric primitives, and typically represents a single object that may be used multiple times in the scene. BLAS objects are constructed from a set of *geometries*, each containing either a triangle mesh or AABB primitives for procedural geometry defined via intersection shaders. The use of multiple geometries also allows the use of different shaders within a single BLAS through the shader table configuration. Once constructed, BLASes may then be used to populate top-level acceleration structures.

Top-Level Acceleration Structures

Top-level acceleration structure (TLAS) objects represent an entire raytracing scene; they do not contain any geometry themselves, instead functioning as a container for one or more BLASes. The TLAS is constructed from a set of BLAS *instances*, each referencing a single BLAS's geometry and

transforming it into the TLAS's coordinate space with a model matrix. Each BLAS may be instantiated multiple times, similar to mesh instancing in the rasterization pipeline. Each instance has several user-provided parameters which affect raytracing behaviour: instance flag bits, which may be used by the raytracing shader to ignore subsets of the geometry at runtime; a shader table offset, which enables certain shader table layout configurations (section 3.2.3); and an instance ID number which may be accessed directly in the shaders with the `InstanceID` intrinsic. TLAS objects are bound to shaders through SRV descriptors and are used through a set of new HLSL language features.

3.2.2 Raytracing Shaders

As a hardware-rendering API, the bulk of DXR's functionality resides in shader programs on the device, introducing a completely new shader pipeline and a set of supporting intrinsic functions for HLSL. The shader stages of the new pipeline model the programmable parts of a typical raytracing framework, enabling user-defined material and camera models, implicit geometry, and background illumination. Raytracing is invoked by shaders with the `TraceRay` HLSL intrinsic function, delegating scene traversal and ray-primitive intersection to fixed-function raytracing hardware. The different shader stages (with the exception of ray generation) act as programmable callback functions invoked by the raytracing runtime, directing control flow and implementing the rendering calculations. Data is passed between stages with a user-defined `RayPayload` structure, which may be used to pass supplementary shader parameters and return shading results (section 3.2.3). Subsequent shader stages may themselves recall `TraceRay`, yielding a recursive execution model that is well-suited to path tracing and PBR.

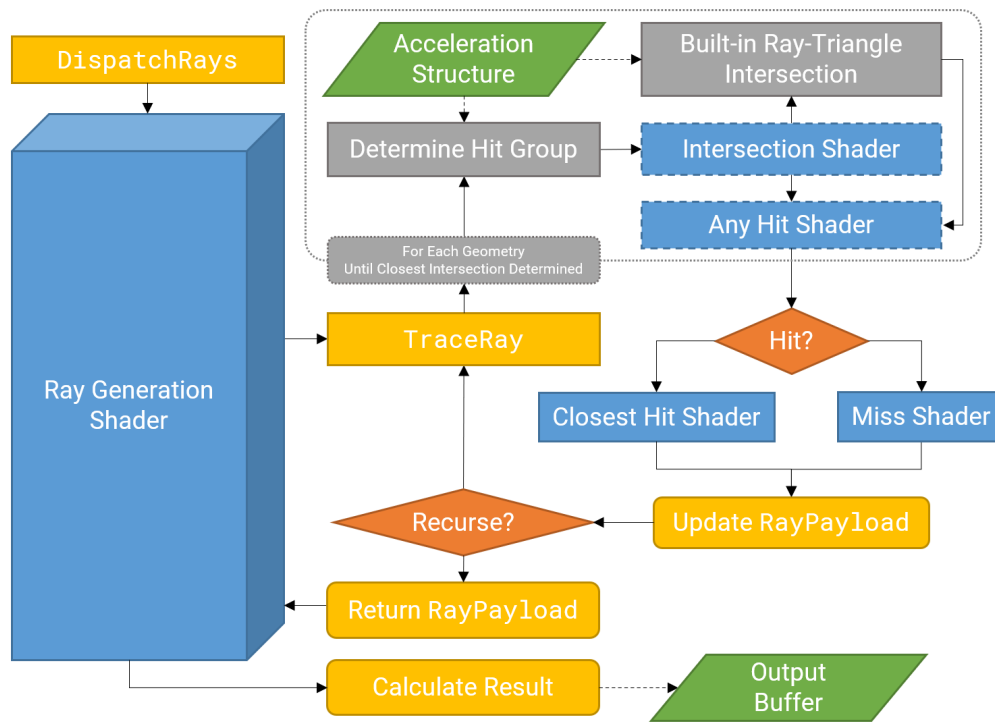


Figure 3.1: Overview of DXR’s shader stages and control flow. The pipeline begins by dispatching a grid of ray generation shaders. Raytracing is then initiated with the `TraceRay` intrinsic, which subsequently calls the hit group shaders associated with any geometry it hits. Closest hit and Miss shaders invoked by `TraceRay` may invoke the procedure again themselves, enabling a recursive execution model. All shaders communicate through a user-defined `RayPayload` structure which can hold shader arguments and return values.

Ray Generation Shader

The **ray generation** shader is the first stage of the raytracing pipeline and the only stage invoked directly by the host. As the name suggests, it generates the primary rays for the renderer and additionally is responsible for accumulating and writing the image results. Unlike pixel shaders, the ray generation shader does not automatically write its output to an API-designated framebuffer; instead, it functions like a compute shader.

Ray generation is dispatched directly by the host in a 1, 2, or 3-dimensional grid, with each shader thread receiving a unique ID corresponding to its grid position. To render the output, a writable buffer must be provided to the shader through a UAV descriptor and then copied to D3D12's framebuffer after raytracing has finished. This allows for flexible usage of the raytracing pipeline beyond rendering a 2-dimensional image, such as volumetric or discrete point sampling.

In a typical rendering setup, the ray generation shader is dispatched with dimensions equal to the image resolution, with each shader generating ray samples according to the camera model. The shader then samples the irradiance of the scene using `TraceRay`, integrates the results and writes to its corresponding pixel in the output buffer.

Closest Hit Shader

The **closest hit shader** is the other core shader stage of the raytracing pipeline. It is invoked by the raytracing runtime at the point of intersection on the nearest surface. This stage is most similar to a pixel shader as it is executed on every visible surface and will typically contain the implementation for material shading. Closest hit shaders may themselves invoke subsequent calls to `TraceRay`, enabling indirect lighting calculations and path tracing.

Unlike pixel shaders, the closest hit shader does not automatically receive interpolated vertex data. Instead, the raytracing runtime provides

some geometric information about the intersection to help the shader implementation fetch or generate the corresponding vertex positions and attributes. This includes the primitive index, barycentric coordinates, and distance along the ray, among others. Because the acceleration structure geometry is opaque, shaders must hold their own copy of vertex data to implement their shading.

Miss Shader

The miss shader is invoked in place of the closest hit shader if the ray query does not intersect the specified geometry. Typically this will be used to render the background of the scene.

Intersection Shader

The intersection shader is an optional shader stage that allows for the implementation of procedurally defined geometry, rather than a triangle mesh. If an intersection shader is provided, the ray query will instead trace against the object's bounding box (specified during the acceleration structure build), and if that intersects, the intersection shader is called. The intersection shader takes the original parameters of the ray query and is responsible for determining if it intersects the geometry within the AABB. This can be used to implement geometry such as spheres with analytical methods, or more complex objects such as height fields and isosurfaces without the need for polygonizing their surface.

Any Hit Shader

The any hit shader is an optional shader stage that functions similarly to the closest hit shader. It is invoked by the runtime on potential candidates for the closest hit and is able to direct the control flow of the runtime to prematurely accept or reject the candidate as the closest hit. The two primary

use cases are to improve efficiency for rendering transparency and shadows - the hit may be ignored to render transparent objects, or accepted if a light source is occluded by any object along the ray. It may also update the payload accordingly. This shader stage makes weak guarantees about its invocation and cannot invoke `TraceRay` itself.

Callable Shaders

Callable shaders are not integrated into the raytracing pipeline by default, but allow general-purpose compute shaders to leverage the dynamic execution capabilities that were introduced with the raytracing pipeline. Callable shaders function very similarly to hit and miss shaders, except they are called directly by the invoking shader rather than indirectly through the raytracing runtime, and may invoke other callable shaders recursively.

3.2.3 Shader Execution Model

To support the dynamic dispatch of shaders, DXR introduces entirely new, dynamic shader and resource binding mechanisms that forgo the up-front binding model of rasterization. At the same time, the new HLSL intrinsics allow the shaders to direct the control flow of the raytracer and query geometric calculations from the hardware raytracing runtime.

Shader Records

DXR extends the DX12's state-driven resource binding API and binds raytracing shaders and their resources through new objects called **shader records**. Shader records are a transparent data structure comprised of two parts: one or more **shader identifiers**, 32-byte blobs which uniquely refer to a compiled shader program on the device; and the instantiated **local root arguments** that are required by the referenced shaders.

As an extension to global root signatures, raytracing shaders now additionally define a **local root signature** to specify resource requirements

that are not applicable at global scope. This enables a high degree of heterogeneity between in-flight shader programs with potentially differing resource requirements, and also allows shader programs to be instantiated multiple times with multiple sets of arguments. In particular, local root arguments may be used to store the specific material properties of objects in the scene, or contain references to associated data such as vertex attribute buffers which must be stored separately from the raytracing acceleration structure.

There are three kinds of shader records corresponding to different stages of the raytracing pipeline:

Ray generation and *miss shader records* each contain a ray generation or miss shader identifier and the shader's local root arguments, and are bound by the runtime whenever either shader stage is executed.

Callable shader records are much the same but are not required for raytracing and are manually selected in the shader rather than automatically by the DXR runtime.

Hit group shader records contain the shader identifiers and local root arguments for a specific **hit group**; a bundle of closest-hit, intersection, and any-hit shaders. The any-hit shader is optional, and the intersection shader is only required for procedural geometry, defaulting to DXR's built-in triangle intersection algorithm if not specified. Hit groups are bound at runtime when `TraceRay` is invoked, assigning all raytracing with a specific hit group shader record to dynamically specify its shading implementation.

Shader Tables

Shader tables form the basis of DXR's dynamic shader dispatch mechanism: linear arrays of shader records that the user creates directly in GPU memory and passes to the API through virtual pointers. Each type of shader record has a corresponding shader table: the *ray generation*, *miss*,

callable, and *hit group shader tables*; although in practice the ray generation shader table only contains a single record and the callable shader table may be omitted. These are then dynamically indexed at runtime, binding and executing the shader record at the specified index in the table. While the contents of each shader record may vary due to differing local root signatures, all entries in each shader table must have a fixed stride to support dynamic addressing by index.

Shader tables must be carefully laid out by the user to match how the shader records are indexed. For miss and callable shaders, shader table indices are supplied directly by the calling shader, requiring only that the user populates the tables in a consistent manner. However, to support dynamic shader dispatch from `TraceRay`, the hit group shader table uses a fixed-function addressing calculation that tightly couples the shader table with acceleration structure layout.

Hit group indices are calculated within the `TraceRay` invocation according to the following pseudocode:

```
hitgroup_index = traceray_offset
                + (blas_geometry_index * traceray_geometry_stride)
                + tlas_instance_offset
```

The caller of `TraceRay` provides two addressing parameters, the first providing an initial offset into the table that can be used to partition it into different sets of functionality that may be required by the shaders. The other parameter from `TraceRay` controls the stride contribution to the hit group index from BLAS geometries; each *geometry* during BLAS construction is assigned a unique index in the order they were provided to the API (section 3.2.1); this allows each geometry within the BLAS to use different shaders to create objects composed of multiple different materials. Finally, each *BLAS instance* in the TLAS contributes a fixed offset that is user-specified during TLAS construction (section 3.2.1).

In a trivial setup, only the instance contribution may be used, mapping each BLAS instance to a single hit group by setting the geometry stride

and table offset to zero. By utilizing all addressing parameters, the instance contribution can be used to allocate variably sized ranges of shader records for each BLAS that are selected through geometries' contribution. This allows for a broad range of addressing schemes that must be coordinated between the shader table and acceleration structure.

Chapter 4

Implementation

In this section, I will describe in detail the implementation of my raytracer. It is implemented in C++ and HLSL using the Windows and DirectX 12 APIs. A small number of open source C++ libraries are used: DirectXMath for vector and matrix arithmetic; Dear ImGui for user interfacing; STB for image capture output; as well as the C standard library.

4.1 System Overview

The raytracer is an interactive, GPU-based renderer that supports progressive rendering of a static scene and real-time adjustment of a wide range of rendering and material parameters. It implements path tracing in DXR that rapidly accumulates image samples to provide immediate user feedback. Its primary feature is first-class support for translucent materials with a physically-based subsurface light transport model, allowing real-time adjustment of scattering and absorption terms under global illumination.

Outside of mesh loading, windowing, and UI, the bulk of the software is implemented with shaders on the GPU. There are three major GPU-driven components: Image sampling (ie. the raytracer); surface irradiance sam-

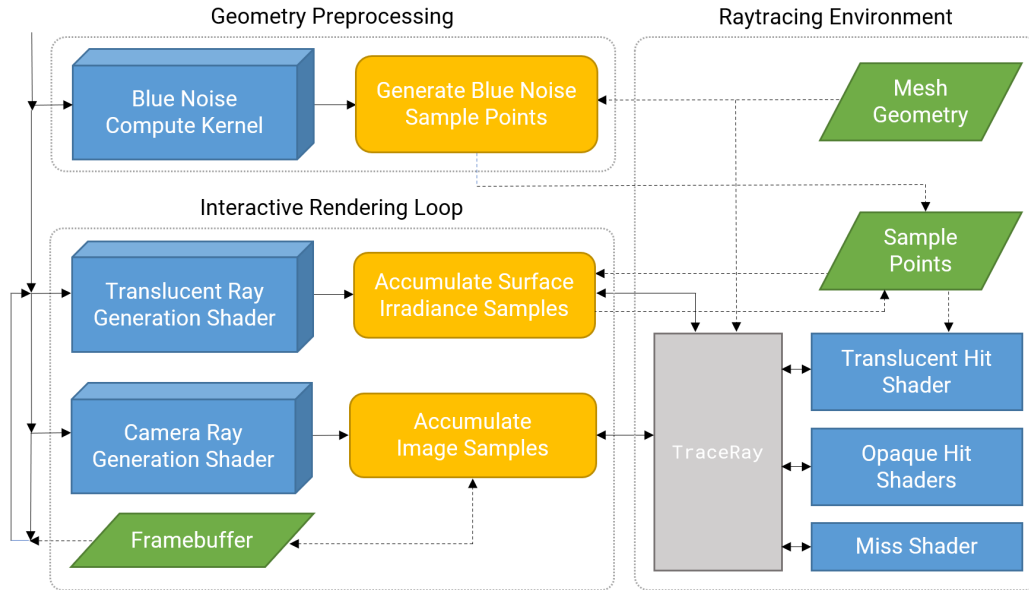


Figure 4.1: GPU pipeline overview. Solid arrows represent control flow and dashed arrows represent data dependencies.

pling for translucent materials; and blue noise surface point generation for evenly distributed irradiance samples.

Both the image sampling and irradiance sampling shaders make use of a shared raytracing pipeline implemented with DXR, while the blue noise point sampling exists as a standalone pipeline implemented with D3D12 compute shaders.

4.2 Software Architecture

My raytracer was developed largely from scratch with minimal dependencies to enable a high degree of flexibility during development. The main programming philosophy that I chose to employ is data-oriented design (as opposed to object-oriented design), which seeks to minimize layers of abstraction and promote *plain-old-data* as the preferred representation of program state. This approach is beneficial for rendering as it reduces the

number of discrete allocations and object graph complexity and simplifies state transfer between host and device.

4.2.1 Modules

In keeping with the data-oriented philosophy, the raytracer's codebase is separated into a small number of modules which are each responsible for a subset of its functionality. Each module encapsulates a subset of the program's global state which is modified through their external API.

The `Device` module is responsible for initialising the D3D12 device context and provides a number of utility functions for creating and using D3D12 objects.

The `Raytracing` module creates and configures the raytracing pipeline, and exposes an API for the rest of the program to specify scene geometry, materials, and acceleration structure layout.

The `Bluenoise` module is a self-contained module implementing hardware-accelerated blue noise surface sampling.

4.2.2 Utilities

To avoid the trappings of object-oriented design the C++ standard template library is not used in the project. Instead, I have developed simple in-house replacements for common workhorse classes emphasising data transparency. The primary utility classes are `ArrayView` and `Array`, replacing STL's `std::slice` and `std::vector` respectively, and are used extensively in the application's internal APIs.

```
template<typename T>
struct ArrayView {
    T*      ptr;
    size_t len;
};

template<typename T>
struct Array
: public ArrayView<T>
{
    size_t cap;
};
```

`ArrayView<void>` is a template specialization that is used by some raw data transfer functions to represent an untyped memory range (i.e. a `void*` plus length tuple) and does not implement functionality beyond stride-checked reinterpret casts.

4.2.3 Execution Model

The rendering context, including window handles, descriptor heaps, command list, and command queue, is managed by the program's `main` function and is passed as needed to modules' API functions. Specifically, modules that require the GPU to submit their operations to the command list but do not execute them themselves - the command list is retained within the main function and executed at an appropriate time after the API function returns.

```
ID3D12Resource* Device::create_buffer_and_write_contents(
    ID3D12GraphicsCommandList* cmd_list,
    ArrayView<void> data,
    /* ... */
);
void Raytracing::dispatch_rays(
    ID3D12GraphicsCommandList4* cmd_list
);
```

This models GPU-facing functions as extended command list operations that also perform their necessary set-up on the CPU side. This approach centralises device synchronisation for the entire application and has good separation of concerns for additional render passes, such GUI rendering with the third-party library Dear ImGui.

One of the main issues this pattern presents is that modules must stage temporary resources on the GPU while waiting for the command list to execute on them. This is commonly required when transferring data between host and device. I employ a simple memory management solu-

tion where the device module maintains a list of temporary resources that modules can submit to, which are then released at the end of every frame after rendering is completed.

```
namespace Device {  
    extern Array<ID3D12Resource*> g_temp_resources;  
  
    void push_temp_resource(ID3D12Resource* resource);  
    void release_temp_resources(); // called once per frame  
}
```

More advanced device memory management strategies are possible with the D3D12 allocation API, however, resource allocation in the renderer is infrequent beyond program initialization, so this simple solution is sufficient.

4.3 Geometry Processing

The first stage of the raytracing pipeline is loading and preprocessing the scene geometry. This makes up the bulk of the program initialization and performs the steps necessary to transform the input geometry and scene description into the format used by the raytracing pipeline.

The raytracer does not yet support scene description files but uses a simple internal API in the source code as an intermediate step towards a human-readable format. Its overall structure mirrors DXR's internal geometry format at a high level, supporting explicit control over BLAS geometries and instancing, while allowing the implementation to manage the dependencies between the acceleration structures and shader tables.

4.3.1 Mesh Loading

Geometry is first loaded into an index format from a set of `.obj` files and subsequently formatted into a set of `GeometryInstance` structures, specifying shader and material properties.

```

void parse_obj_file(
    const char* filename, bool convert_to_rhs,
    Array<Vertex>* vertices,
    Array<Index>* indices
);

namespace Raytracing {
    struct GeometryInstance {
        ArrayView<Vertex> vertices;
        ArrayView<Index> indices;
        Material material;
    };
}

```

Each `GeometryInstance` corresponds to a *geometry* within a BLAS in DXR's API, with each geometry being assigned a single entry in the shader table. Three shader types are currently supported: `Lambert` for diffuse surfaces; `Light` for volume lights; and `Translucent`, which covers a broad range of subsurface scattering properties.

4.3.2 BLAS Construction

The `GeometryInstance` structures are then passed into the `build_blas` function. This uploads the mesh data to the GPU, constructs the DXR BLAS object from the set of geometries, and assigns it a region of the shader table.

```

namespace Raytracing {
    struct Blas {
        ID3D12Resource* blas;
        ID3D12Resource* vb;
        ID3D12Resource* ib;

        UINT shader_table_index;
        UINT translucent_ids_index;
        UINT translucent_ids_count;
    };
}

```

```

Blas build_blas(
    ID3D12GraphicsCommandList* cmd_list,
    ArrayView<GeometryInstance> geometries
);
}

```

Each BLAS represents a single object or set of objects that may be instanced and placed into the scene. The `Raytracing::Blas` structure contains references to the D3D12 resources containing the BLAS object, the GPU's copy of the index and vertex buffers, and metadata that is used to reference the BLAS's corresponding shader table slots (section ??).

The `Raytracing` module internally maintains and updates the global shader table. Each new BLAS is sequentially assigned a region of shader table slots for each of its geometries, to be shared by all instances of it in the TLAS.

The vertex data from all geometries is concatenated into a single pair of vertex and index buffers to reduce the number of GPU objects. Each geometry stores an offset into the index buffer in its shader record that is then used to query its specific vertex data in the shader.

This procedure also performs initial bookkeeping for translucent geometries, each requiring an additional set of resources for their irradiance samples. Every BLAS geometry with the `Translucent` shader is sequentially assigned a unique `translucent_id`. This is used by the `Raytracing` module to generate blue noise surface samples and to reference them in the shader.

4.3.3 TLAS Construction

The TLAS is constructed with the `build_tlas` function, specifying the layout of the scene. It takes a set of `BlasInstance` structures that pair an object-to-world-space transform with a BLAS reference, allowing multiple instantiations of each BLAS. This builds the DXR TLAS object, uploads the

finalised shader table, and generates an initial set of unique set of irradiance sample points for all translucent geometry instances (section 4.6).

```
namespace Raytracing {
    struct BlasInstance {
        XMFLOAT4X4 transform;
        Blas* blas;
    };

    void build_tlas(
        ID3D12GraphicsCommandList4* cmd_list,
        ArrayView<BlasInstance> instances
    );
}
```

The resulting TLAS is stored internally in the `Raytracing` module and used as the scene for the raytracer.

Each BLAS instance within the TLAS build is assigned its shader table offset

The `transform` and the `Blas`'s BLAS resource and metadata are used to populate the `D3D12_RAYTRACING_INSTANCE_DESC` structures that are required for the top-level acceleration structure build.

4.4 Raytracing Procedure

4.4.1 Path Tracing

The raytracer uses a simple implementation of path tracing that directly integrates individual light paths through the scene. While DXR enables recursive calls to `TraceRay` in hit and miss shaders, each recursive call requires stack space that must be declared up-front that should be minimised.

My implementation uses a non-recursive model that only samples a single direction on the hemisphere for each bounce, unlike the typical approach which integrates multiple samples, requiring a maximum recur-

sion depth of 1. This initially produces noisier results and converges more slowly but is an equivalent form of the path integral LTE that significantly reduces computational complexity per primary ray.

The path sampling procedure takes advantage of linear terms in the rendering equation to iteratively step along path segments and accumulate the partial reflected radiance (due to emission) and partial attenuation (due to surface reflection) with a fixed number of variables. Given:

$$L = L_e^1 + R^1 (L_e^2 + R^2 (L_e^3 + R^3 (\dots R^{N-1} \cdot L_e^N)))$$

For a path with N bounces, the total transmitted radiance along a path can be expressed in terms of an iterated sum and iterated product:

$$L = \sum_{i=1}^N \left(L_e^i \cdot \prod_{j=1}^{i-1} R^j \right)$$

This is implemented in the `trace_path_sample` procedure with the following pseudocode:

```
trace_path_sample(ray, bounces):
    transmission := 1 // attenuation along path
    radiance      := 0 // accumulated emission

    // iterate over path segments
    for i := 0..bounces:
        payload := TraceRay(ray)

        // integrate reflectance and emission from path segment
        radiance += reflectance * payload.emission
        transmission *= payload.reflectance

        // step ray to next point on path
        ray.origin += ray.direction * payload.distance
        ray.direction = payload.scatter_direction

    return radiance
```

The procedure is tightly coupled with the closest hit shader implementations, which are responsible for returning the `RayPayload` that specifies how materials are rendered.

4.4.2 Hit Shaders

Closest hit shaders and the path sampling procedure communicate through the user-defined `RayPayload` structure. This is passed as an `inout` variable to the `TraceRay` intrinsic, allowing two-way communication between the stages.

```
struct RayPayload {  
    uint    rng;  
    float   t;  
    float3  scatter;  
    float3  reflectance;  
    float3  emission;  
};
```

The `rng` field contains the random number generator state which is seeded per-sample-per-frame by the ray generation shader. The modified value is returned by hit shaders to ensure all hit shader invocations receive a unique random sequence. Random number generation is implemented with a 32-bit Xorshift PRNG[11].

The `t` field contains the distance travelled by the ray to intersect the hit surface. This is available to the hit shader through the `RayTCurrent` intrinsic, but must be returned to the hit shader for it to calculate the ray's point of intersection.

`scatter`, `reflectance`, and `emission` collectively specify the material properties of the hit surface. `scatter` returns a random direction on the hemisphere of the surface that was hit. This allows closest hit shaders to individually determine their sampling distribution, such as a cosine-weighted distribution for Lambertian surfaces or perfect reflection for mirror-like surfaces. `reflectance` then returns the 3-channel BRDF

for the incident ray (via the `WorldRayDirection` intrinsic) and the sampled direction. This must also be normalized if non-uniform sampling is used. Finally, `emission` returns the 3-channel emissive component of the material. This is used for light sources, which do not reflect light, and translucent materials, which combine surface reflections and non-local subsurface light transport.

4.5 Blue Noise Sample Point Generation

Blue noise sample points are generated through a separate compute pipeline in the `Bluenoise` module. The module implements a slightly modified version of Bowers' parallel algorithm[2], splitting it into preprocess and point-generation steps to enable rapid resampling of surfaces in the interactive renderer. This is extremely beneficial for translucent material tuning as the required sample point density is directly correlated to the mean free path of the BSSRDF. It is also extended to support (angle-preserving) 3-dimensional transformations of the source geometry, allowing multiple instantiations of the same geometry in the raytracer to share their base mesh and preprocess data.

The algorithm works by first creating an initial dense set of uniform random samples on the surface of the mesh, then selecting samples from the initial set that are a minimum distance away from all other selected points, yielding a blue noise distribution. The key observation that allows this algorithm to be executed efficiently on the GPU is that points may be drawn in parallel if the distance between them is known to be greater than the sample rejection radius. This leads to a grid-based algorithm, dividing the geometry into cubic cells of width $r/\sqrt{3}$, and associating each point in the initial set of random samples with its containing cell. Finally, grid cells iteratively draw points from the initial set in parallel *phase groups*, checking adjacent and committing the point if its rejection radius does not overlap

any other committed points.

4.5.1 Mesh Preprocessing and Instancing

In order to mirror the functionality of BLAS instancing provided by DXR and the raytracer's geometry API, the `Bluenoise` module factors the algorithm into separate preprocessing and point generation procedures. Preprocessing the mesh data allows on-demand surface resampling without copying any resources to the GPU, while the point generation procedure modifies the algorithm to support 3-dimensional instance transforms while respecting the sample rejection radius in world-space units.

The preprocessing procedure is the only scalar, CPU-driven operation of the blue noise pipeline, computing an array of cumulative surface areas of each primitive in the mesh. This is uploaded to the GPU to be used by the initial random sample generation stage of the pipeline. It also calculates the mesh's bounding box which is subsequently used to determine the cell grid dimensions. The results are stored in a `Bluenoise::Preprocess` structure that is associated with each translucent geometry.

```
namespace Bluenoise {
    struct Preprocess {
        UINT indices_count;

        Aabb                aabb;
        float                total_surface_area;
        ID3D12Resource*      partial_surface_areas;
    };

    Preprocess preprocess_mesh_data(
        ID3D12GraphicsCommandList* cmd_list,
        ArrayView<Vertex> vertices,
        ArrayView<Index>  indices
    );
};
```


The sample point generation procedure is then called for all instances of the translucent geometry, taking the preprocess data shared between all instances and the instance-specific world-space transform. Mesh data is borrowed from the raytracing environment, whose format concatenates all geometry in each BLAS into a single pair of vertex and index buffers; the mesh data for the specific translucent geometry is accessed through an offset into the index buffer.

```
namespace Bluenoise {
    UINT generate_sample_points(
        ID3D12Resource** sample_points_buffer,
        ID3D12Resource** point_normals_buffer,
        float* scale_factor,

        Preprocess* preprocess,
        ID3D12Resource* ib, UINT ib_offset,
        ID3D12Resource* vb,
        XMFLLOAT4X4* transform,
        float rejection_radius
    );
}
```

To account for instance scaling, the uniform scale factor is determined from the transformation matrix and returned to the caller; non-uniform scaling cannot be supported as this would require completely recalculating the partial surface areas array. The world-space rejection radius is then transformed into model space by inversely scaling it, giving $r_{\text{local}} = r_{\text{global}}/s$. This adjusts the algorithm to sample more or less densely according to the scaling of the instance. Translation and rotation do not need to be accounted for at this stage as they do not affect surface area.

Finally, as blue noise samples are drawn they are converted into world space by applying the transform matrix. Because each instance experiences different lighting conditions it is not possible to have a shared set of points between instances; they are transformed up front to reduce the code complexity of the translucent rendering shaders.

4.5.2 Initial Random Point Generation

Generating the initial uniform random samples is performed in two steps: First, cumulative surface areas of each primitive in the mesh geometry are calculated and stored in an array; this is done ahead of time by the preprocessing procedure. Then, primitives are then randomly selected from this array, weighted by surface area, and a random point is generated on the triangle and stored in the initial set.

Generating the initial sample points is trivially parallel and is performed on the GPU in a single compute kernel. The procedure allocates a fixed-size array for the initial points and dispatches a shader thread to generate each element.

Accurately estimating the required size of this set is important for generating good-quality results. An upper bound on the number of blue noise samples can be calculated by assuming each point occupies a circular region of half the rejection radius: $N \leq \lfloor A / (\pi(\frac{r}{2})^2) \rfloor$. My implementation generates initial sample points with a factor of 16 over this upper bound, rounded to the next power of two to simplify subsequent compute kernels. Increasing the factor beyond this did not noticeably improve visual results.

4.5.3 Hashtable Construction

After the initial sample points are generated they are associated with their cells through a hashtable. A hashtable is used rather than a volumetric grid as typically only a small proportion of the cells will lie on the object's surface. The hashtable is constructed by first sorting the initial points array by cell; this partitions the set into linear arrays of all the points within each cell. The hashtable entry for each cell is then assigned the first index of its partition, functionally mapping all occupied cell IDs to a list of random points within the cell.

Each bucket in the hashtable also contains a member to store the selected sample point from each cell. This is initialised to a default value and is used by the final kernel to reject sample points that are too close.

Sorting is implemented with a parallel bitonic sort kernel. The kernel is dispatched iteratively over the array, and sorts the points in a fixed number of iterations by comparing and swapping pairs of elements according to a predefined pattern.

The hashtable is implemented as a simple bucketed hashtable. The indices of each partition of the sorted buffer are found by dispatching a kernel on each element and checking if its predecessor belongs to a different grid cell. The indices are then added to the hashtable in parallel, using an atomic add instruction to allocate each cell ID a unique bucket in the case of hash collisions. My implementation allocates 5 buckets for each table entry. Due to the complexity of implementing probing on the GPU, my implementation discards entries in the case of overflow, however, this situation is uncommon in practice.

4.5.4 Sample Point Selection

The final stage iteratively draws samples from the cell grid in parallel *phase groups*. Each cell in the phase group selects a single trial point from the initial random set, testing it against the selected points (if present) in all adjacent cells. If the trial point does not lie within the rejection radius of any other selected points, it is added to the set of blue noise samples and is stored in the cell's hashtable bucket. This process is executed for each phase group in a randomly determined order, and is repeated until all points have either been rejected or selected.

To ensure parallel threads do not select conflicting points, the phase groups must be separated by a minimum of 1 cell width for a minimum

number of $2 \times 2 \times 2$ groups; in my implementation I use $3 \times 3 \times 3$, as recommended in the paper.

After the final set sample points has been calculated it is passed back to the `Raytracing` module for irradiance sample collection and BSSRDF integration.

4.6 Diffuse Reflectance Evaluation

After the blue noise sample points have been generated, evaluating diffuse reflectance for translucent geometry is implemented in two steps: collecting irradiance samples and integrating them with the BSSRDF. This approach is based on the work of Jensen and Buhler in their 2002 paper[8]. Both stages must consider a range of optical phenomena to produce accurate results and maintain conservation of energy within the scene.

4.6.1 Irradiance Sampling

First, surface irradiance is sampled by tracing path samples from the surface of the translucent geometry. Each blue noise sample point has its surface normal stored in a separate buffer that is used by this stage. The ray generation shader is dispatched for every sample point in the scene and path samples are drawn randomly from the hemisphere of the surface.

The incident irradiance contribution from the path sample is then calculated using the cosine law. This is separated into a transmitted component and a reflected component using a Fresnel term; the reflected component is discarded. Finally, the contribution is stored in the sample point's payload. The results are averaged by sample point area and accumulated over time.

To allow both stages to run in parallel, the irradiance sample collection writes its results to a clone of the sample points buffer used by the BSSRDF integration. The updated sample points of the former stage are then

copied to the latter stage once per frame.

4.6.2 BSSRDF Integration

Integrating the BSSRDF is done on demand in the translucent material hit shader. This trivially implements the surface integral by iterating over all sample points on the geometry and calculating their contribution with the BSSRDF. In addition to emission due to subsurface scattering, this stage also calculates surface reflectance.

The BSSRDF is computed in RGB space using Jensen et al.'s dipole diffusion model[9].

Chapter 5

Results

Development of the raytracer was done on a Windows 10 PC with a 6-core Intel Xeon processor and an Nvidia RTX 2080 Ti GPU. The application (fig. 5.1) runs in an interactive sample accumulation loop, allowing real-time adjustment of camera and material parameters. The image is reset whenever parameters are modified and rapidly begins accumulating samples, giving instant feedback to changes.

It supports a single translucent material, whose properties can be adjusted at runtime, and is applied to all translucent objects in the scene. The translucent geometries each store a set of surface irradiance sample points that accumulate in tandem with the image, which are similarly reset when the lighting conditions or the surface transmission properties are altered.

Before implementing translucent materials, the first stage of development was to lay the foundations of the path tracer and render a basic scene. I have used a slightly modified Cornell Box scene (fig. 5.2) through most of development, as it provides a simple test bed that is easily adapted to test different shaders. It's also a good fit for translucent materials due to having a large, single light source showcasing the material's diffusion profile at two different heights.

With the basic pipeline in place, implementing full support for translucent material would be the main focus of the development past this point.

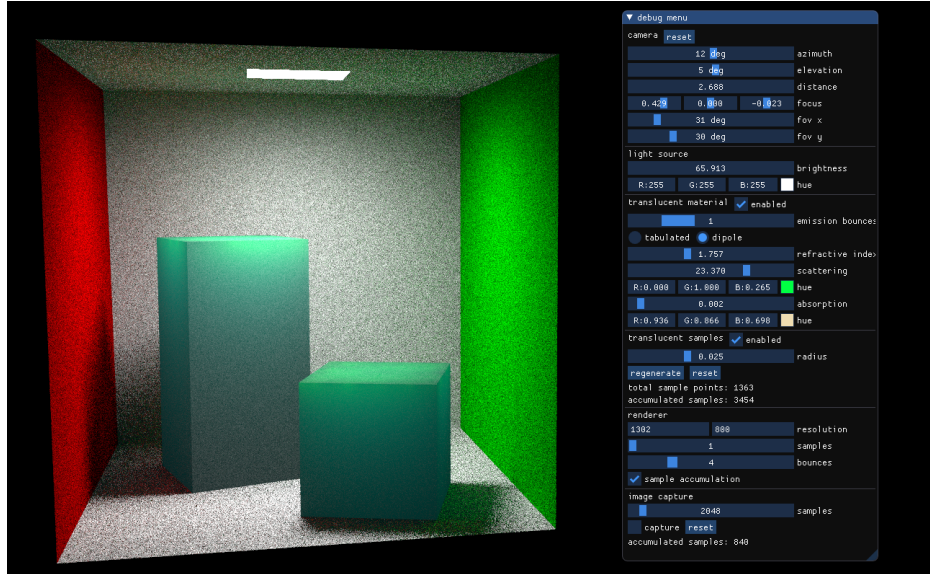


Figure 5.1: Screenshot of the final application, showing a partially rendered scene and parameter controls.

This is a synthesis of multiple complex features that had to be broken into multiple steps. I first implemented the BSSRDF model on static geometry that I could trivially generate a uniform set of sample points on, as well as the initial versions of the shaders for irradiance sampling and integration (fig. 5.3).

The model is very sensitive to sample point density due to the shader integrating the BSSRDF over a discrete set of points. If the mean free path of the material is significantly lower than the distance between a set of points then they appear as visibly bright patches on the object's surface.

This limits the range of materials that can accurately be displayed by the model based on the maximum point density. Recall that the mean free path is given by the reciprocal of the extinction coefficient: $1/\sigma_t = 1/(\sigma_a + \sigma_s)$. This is problematic: for the mean distance between sample points to maintain parity with the mean free path, the average density increases with the square of the extinction coefficient, causing the model to

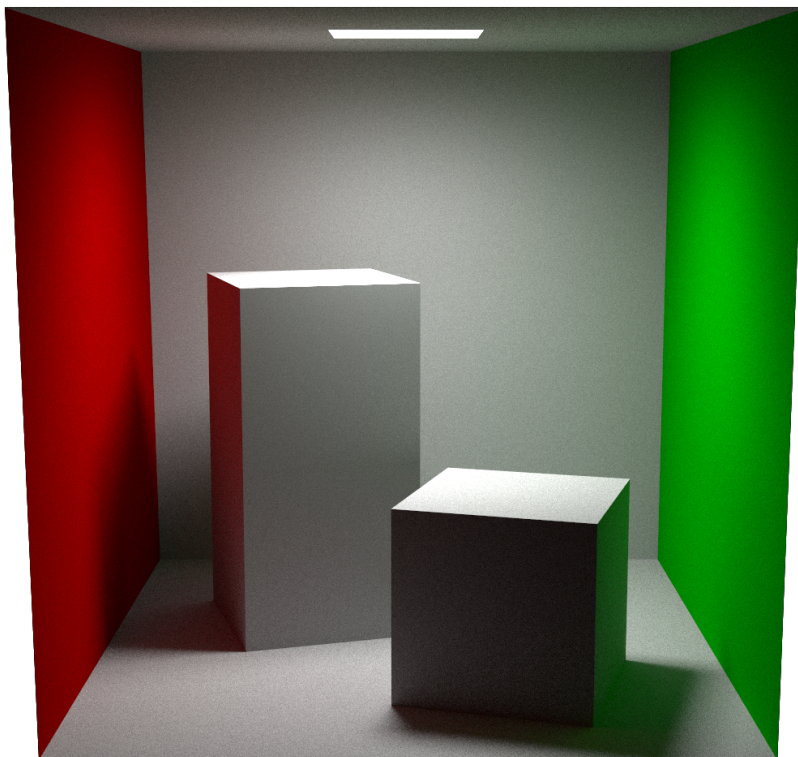


Figure 5.2: The requisite Cornell Box render, demonstrating an emissive surface, soft shadows, and diffuse interreflection.

rapidly approach an upper bound on memory and computation for highly scattering or absorbing media.

While the blue noise algorithm is readily capable of quickly generating a very high density of samples, there is a hard limit around $\sigma_t < 50$ in the renderer (variable depending on surface area of translucent objects) due to the aforementioned computational complexity bound. I have specifically chosen to show low-extinction media in the results for that reason; I discuss possible solutions in the potential expansions section (6.1.3). In any case, this is a better representation of the nonlocal capabilities of the renderer, as high-extinction media tends towards being opaque appearance.



Figure 5.3: Earliest BSSRDF implementation using a regular grid of sample points. The visible sample points are due to the material having a much lower mean free path than the distance between the samples. Dark patches were due to a bug. The rightmost cube demonstrates a subtle but well-formed diffusion profile.

The GPU blue noise sample point generation algorithm is implemented in a separate compute shader pipeline. It is able to generate very large numbers of points with a very fast response time, enabling live tuning of the sample point density to match the requirements of the translucent material (fig. 5.4, 5.5). Sample point generation is well-integrated with the raytracing pipeline, respecting instantiations of translucent geometry. This modifies Bowers’ algorithm[2], splitting it into a preprocessing step that caches transform-invariant intermediate results, and extending the sample point generation step to convert the world-space rejection radius into local space. This accounts for instance translation, rotation, and scaling, transforming the generated samples into world-space as they are generated. After fully integrating the pipeline I first experimented with single-channel BSSRDFs, with uniform scattering and absorption cross-sections across the spectrum (fig. 5.6, 5.7). As previously mentioned, the renderer is best suited to low-extinction media, producing very appealing results with a large mean free path. These wide diffusion profiles give a very convincing impression of volumetric illumination within the object.

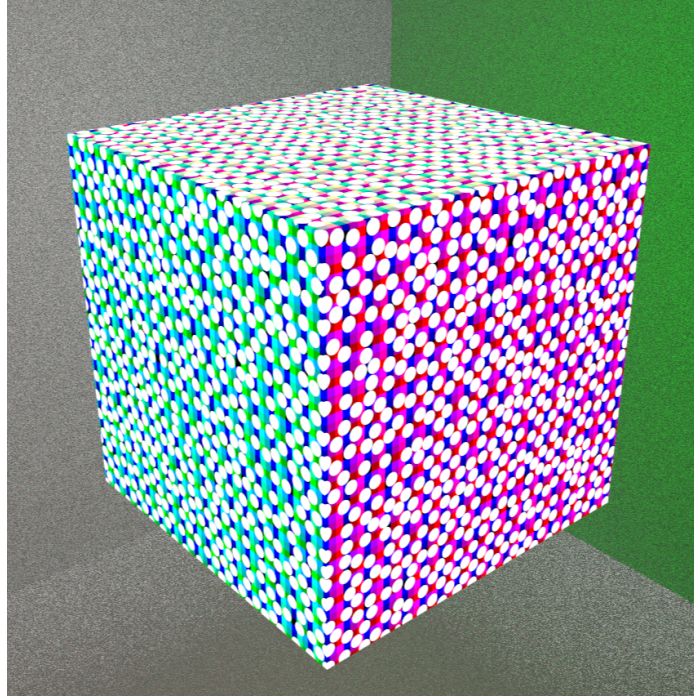


Figure 5.4: Debug visualisation of generated sample points overlaid on their cell grid. Colours represent the $3 \times 3 \times 3$ phase groups. Points are shown with radius of half the rejection radius, representing the largest non-overlapping region around them. While the rejection radius ensures a minimum distance between points, the mean is generally higher. Gaps with the area of the rejection radius are inevitable due to the greedy constraint resolution algorithm on the initial random samples. Axis-aligned planes (as pictured) are a particularly pathological case as they prevent orthogonally adjacent grid cells (width $r/\sqrt{3}$) from mutually selecting sample points, producing somewhat crystalline, periodic regions.

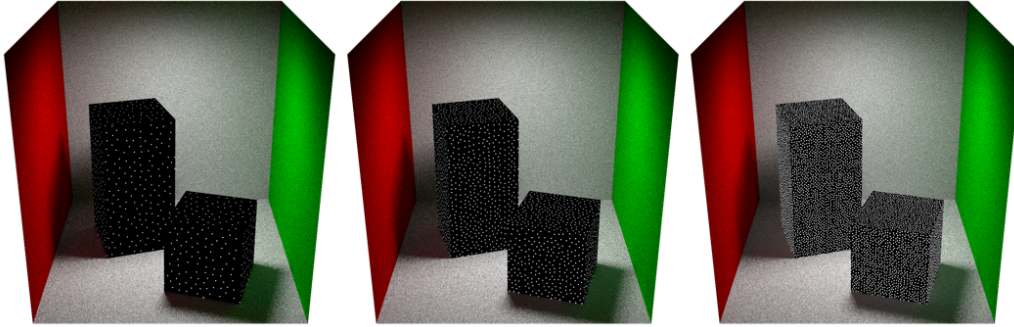


Figure 5.5: Visualisation of blue noise sample points at various densities. From left to right: ($r = 0.025, N = 1330$); ($r = 0.015, N = 3634$); ($r = 0.008, N = 13511$). The total number of points is inversely proportional to the square of the rejection radius, requiring drastically many more points for media with a low mean free path.

Extending the BSSRDF model to support multiple channels was fairly simple and I began experimenting with skin-like rendering. I used a base set of RGB scattering and absorption coefficients approximating skin published by Jensen[9]. I then modified these base values to simulate blood and melanin chromophore concentrations: for blood I increased scattering of the red channel, and for melanin I increased absorption across all channels. While this is a fairly crude approximation it produced a convincing proof of concept for integration with biophysically based BSSRDF models (fig. 5.8).

During development, I experimented with using exported diffuse reflectance data exported from a Monte Carlo skin simulation[5]. The ray-tracer is able to support texture-based BSSRDFs, but it proved difficult to integrate the spectral representation of the data into the pipeline. Physically accurate skin rendering requires taking into account multiple chromophores' spectral profiles resulting in very subtle shifts in hue that cannot be replicated by an RGB model. Implementing a spectral renderer in DXR was well beyond the scope of this project due to its significantly in-

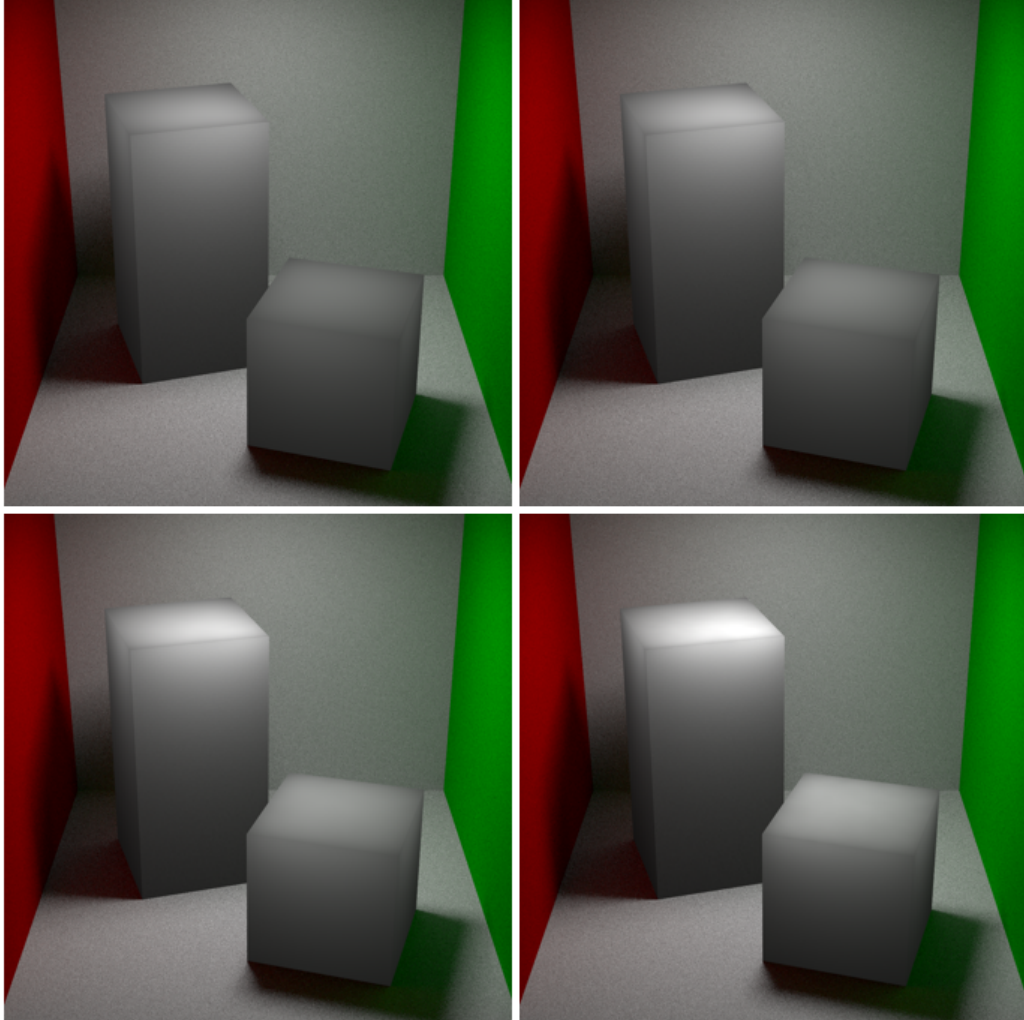


Figure 5.6: Increasing scattering coefficient. From top left to bottom right: $\sigma_s = 10.0, 12.5, 20.0, 30.0$; Absorption fixed at $\sigma_a = 0.005$. These values present large changes in the mean free path, allowing light to be transmitted much more freely through the medium at the lower end, and much less at the higher end. This makes the material appear more opaque and increases the intensity of its brightest spots due to the narrower distribution of scattered light.

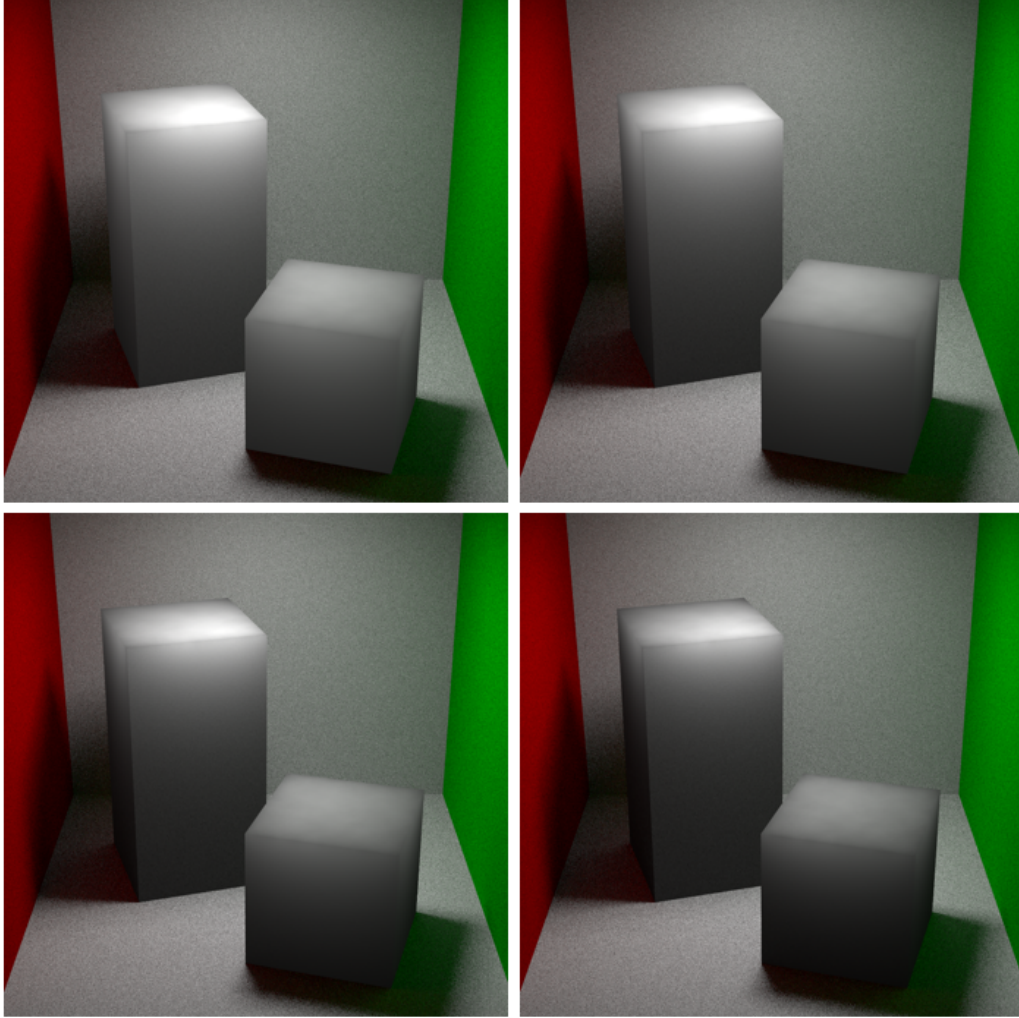


Figure 5.7: Increasing absorption coefficient. From top left to bottom right: $\sigma_a = 0.01, 0.1, 0.5, 1.0$; Scattering fixed at $\sigma_s = 30.0$. The relatively small absorption values have minimal impact on the mean free path, so the radius of the brightest area is visually unchanged. However, transmission is reduced throughout, making a noticeable impact on the brightness of the object, especially at the edge of the diffusion profile.

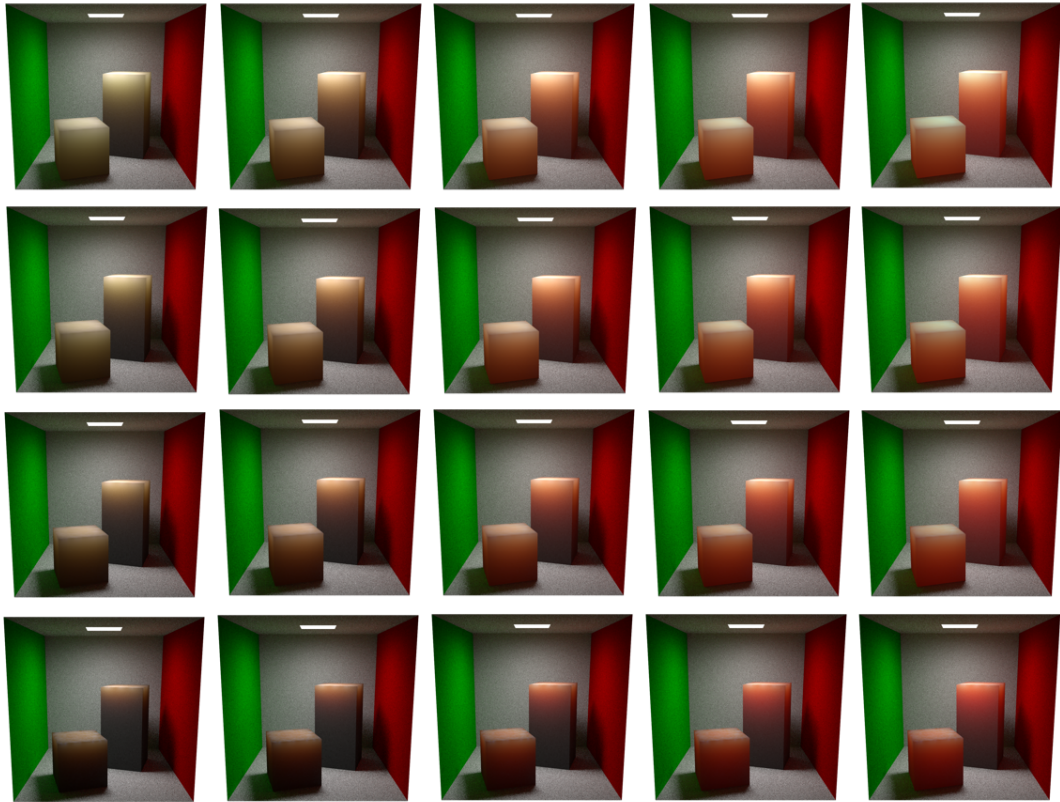


Figure 5.8: Using 3-channel subsurface scattering to imitate the subsurface scattering of different skin compositions. Left-right: blood concentration; top-bottom: melanin concentration.

creased resource requirements. As such this is left on the table for further research, for which I discuss potential solutions in the potential expansions section (6.1.5).

Finally, I applied 3 variants of the skin BSSRDF approximation to a 3D human face scan (fig. 5.9). For this model, I used a much higher density of sample points (10547 in total) than previous renders to support a relatively low mean free path. The results demonstrate the non-local subsurface scattering very well, particularly in and around the nostrils where the thin boundary allows a significant amount of light transmission.

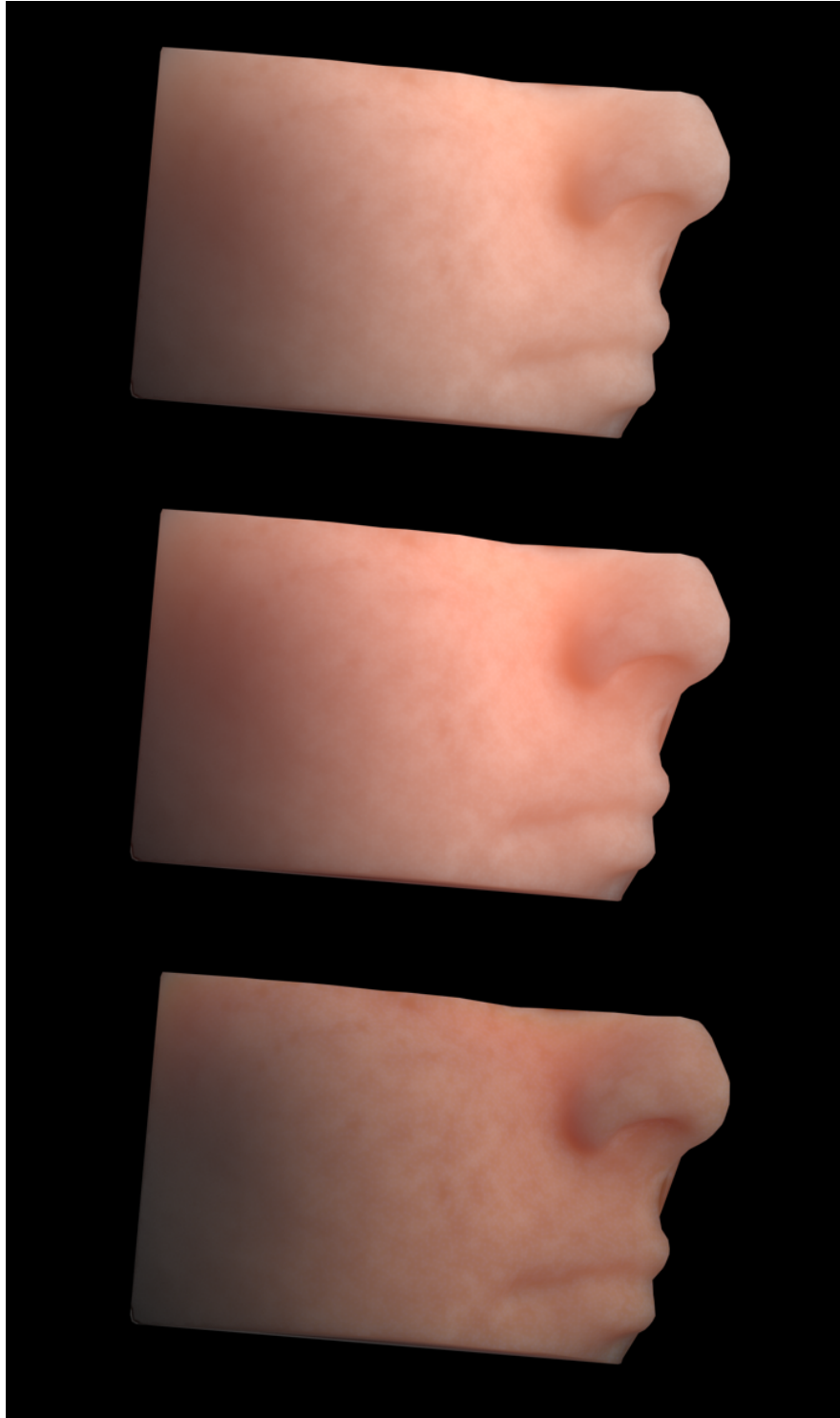


Figure 5.9: Skin BSSRDF applied to 3D facial scan. Top: Low melanin, low blood; Middle: Low melanin, higher blood; Bottom: Higher melanin, higher blood. Model supplied by owner with permission.

In the low melanin, high blood model the hue shifting caused by the non-uniform scattering coefficients is especially visible. On all models, the lighting is diffused smoothly across the cheek despite the light source shining directly downwards. While the lips would benefit from locally increased chromophore saturation, the darkened cleft in the parting is also a nice demonstration of diffuse light from the light-facing surface of the lips partially illuminating the occluded region.

All three models would benefit significantly from a microfacet BRDF model for surface reflections, especially the higher melanin model where specular reflections have higher contrast. They also all suffer slightly from a subtly mottled texture. This is due to pushing the mean free path slightly lower than the sampling density can support, as well as gaps in the distribution due to the exhaustion of the initial random sample set in the sample generation procedure.

On the whole, these renders are an excellent showcase of the translucent material implementation and show promise for future developments of the technology.

Chapter 6

Conclusion and Future Work

I have developed a general-purpose GPU path tracer, utilising modern hardware and APIs, that supports real-time rendering of subsurface scattering under global illumination. I have implemented an extendable pipeline architecture in DXR that can support a wide range of material models and sampling methods. In support I have created a minimal scene creation API that exposes the low-level configuration of acceleration structures provided by DXR while internally maintaining intricately coupled associated data structures.

The raytracer is extended with first-class support for translucent materials to allow real-time visualisation of subsurface scattering properties. I implemented a multichannel BSSRDF model introduced by Jensen et al.[9] that is capable of rendering a broad range of subsurface scattering materials and properties. This is integrated under a two-stage pipeline based on the work by Jensen and Buhler[8] that measures surface irradiance at pre-generated sample points on the geometry and calculates diffuse reflectance by integrating over these. Finally, this is supported by a GPU blue noise surface sampling algorithm based on the work of Bowers et al.[2], capable of near-instantaneous resampling to meet the sample resolution requirements of different BSSRDFs.

6.1 Potential Expansions

6.1.1 Software Features

While the raytracer supports a wide range of parameters, it is currently limited to a static scene that is specified in the source code. While the internal API makes it easy to edit the scene in source, implementing a scene description format is necessary for shipping a useful piece of software. Live scene editing could also be implemented by leveraging modifiable acceleration structures in DXR; this would enable full customisation of the scene while rendering it in real-time.

6.1.2 Physically-Based BRDF Models

As the focus of the renderer was on subsurface scattering, currently it only supports Lambertian reflectance models and has no support for texture mapping. While useful in general, these features are both particularly important for the rendering of realistic skin. Both surface reflectance and subsurface scattering are spatially varying over the body with numerous parameters including roughness, moisture, and chromophore concentrations[6].

6.1.3 Accelerated Irradiance Sample Integration

The current diffuse reflectance integration shader performs an iteration over the entire set of samples for each closest hit shader invocation. While raytracing performance is adequate most of the time, the computation time becomes prohibitive at high numbers of sample points, effectively putting a limit on the highest scattering cross-section that can be rendered in the raytracer. Jensen and Buhler[8] suggest using an octree data structure to hierarchically store irradiance values over areas of multiple sample points. This approach works well for static scenes, but presents challenges with real-time accumulation on the GPU, due to the strong ordering con-

straints of propagating dynamic sample point estimates up through the tree. Interpolation of irradiance estimates between sample points would also improve performance by reducing the required sampling density. Currently, when the surface is undersampled, each point creates a local “bright” spot due to the mean free path being significantly lower than the distance between samples. Efficiently implementing interpolation would likely require some kind of adjacency data for the sample points, which could possibly be efficiently computed by further extending the implementation of Bower’s algorithm[2]. Additionally, the radial symmetry of the BSSRDF models could possibly be leveraged to give a very efficient integral over the region of interpolated irradiance measurements.

Perhaps the most pragmatic option would be to have a separate subsurface scattering model for high-extinction media and apply a hybrid approach. While the required density of sample points increases with the square of the extinction coefficient, the area of non-local reflectance contributions reduces at the inverse rate. This makes it much more feasible to dynamically generate irradiance samples in the neighbourhood of shaded points based on a simple surface parameterisation. This approach would set a general upper bound on the required sample point density, switching to the dynamic sampling model when the mean free path passes this threshold.

6.1.4 Biophysical BSSRDF

To achieve realistic skin rendering a biophysically-based subsurface scattering model must be used. This is implemented using Monte-Carlo methods to accurately simulate the complex layered structure of skin. Integration with a CUDA-based model[5] to generate tabulated BSSRDFs would allow real-time rendering and parameter tuning of realistic skin models.

6.1.5 Spectral Rendering

Spectral rendering is a primary feature of other renderers such as PBRT[12] and Mitsuba[7] and is essential for photorealistic rendering of complex materials. In these renderers, the sample format is extended to a fixed-length array representing samples of the SPD over a fixed range of wavelengths.

This code transformation is trivial and functions efficiently on the CPU, but introduces *significant* complexity for DXR. While typical operations in spectral rendering are linearly independent, which is well-suited to parallelisation, the SPD representations are too large to store in the `RayPayload`. There are several potential ways to get around this limitation, with trade-offs. An initial option is to store SPD samples in global resources while raytracing is in flight. This would require minimal changes to the existing pipeline, but requires a very large resource allocation or a method of allocating and reusing SPD buffers for active shader threads. Due to the current implementation only using a single shader thread per path sample, this would also require iterating vertically through the SPD for vector operations. The global memory overhead could be avoided by assigning each shader thread a limited range of spectra to calculate. This could be implemented by progressively accumulating different spectral ranges, or the shader dispatch dimensions could be increased from 2 to 3. This would increase the total number of shader invocations but allow temporaries to be stored in shader registers and the `RayPayload`, and also enable efficient vector operations on the SPD components.

Finally, there is a possibility to use shader group shared memory to great effect. If large enough, it could possibly store an entire SPD sample, while also allowing a single shader thread to perform `TraceRay` queries and propagate the results to the other shaders working on it. This is a hypothetical feature with several avenues for exploration and will require significant further research and development to bear fruit.

Bibliography

- [1] BLINN, J. F. Models of light reflection for computer synthesized pictures. In Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques (New York, NY, USA, 1977), SIGGRAPH '77, Association for Computing Machinery, p. 192–198.
- [2] BOWERS, J., WANG, R., WEI, L.-Y., AND MALETZ, D. Parallel poisson disk sampling with spectrum analysis on surfaces. ACM Trans. Graph. 29, 6 (dec 2010).
- [3] BURLEY, B. Physically-based shading at disney.
- [4] COOK, R. L., AND TORRANCE, K. E. A reflectance model for computer graphics. ACM Trans. Graph. 1, 1 (jan 1982), 7–24.
- [5] DORONIN, A. The unified monte carlo model of photon migration in scattering tissue-like media for the needs of biomedical optics.
- [6] IGLESIAS-GUITIAN, J. A., ALIAGA, C., JARABO, A., AND GUTIERREZ, D. A biophysically-based model of the optical properties of skin aging. Computer Graphics Forum 34 (2015).
- [7] JAKOB, W., SPEIERER, S., ROUSSEL, N., NIMIER-DAVID, M., VICINI, D., ZELTNER, T., NICOLET, B., CRESPO, M., LEROY, V., AND ZHANG, Z. Mitsuba 3 renderer, 2022. <https://mitsuba-renderer.org>.
- [8] JENSEN, H. W., AND BUHLER, J. A rapid hierarchical rendering technique for translucent materials. In Proceedings of the 29th Annual

- Conference on Computer Graphics and Interactive Techniques (New York, NY, USA, 2002), SIGGRAPH '02, Association for Computing Machinery, p. 576–581.
- [9] JENSEN, H. W., MARSCHNER, S. R., LEVOY, M., AND HANRAHAN, P. A practical model for subsurface light transport. In Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (New York, NY, USA, 2001), SIGGRAPH '01, Association for Computing Machinery, p. 511–518.
- [10] KAJIYA, J. T. The rendering equation. In Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques (New York, NY, USA, 1986), SIGGRAPH '86, Association for Computing Machinery, p. 143–150.
- [11] MARSAGLIA, G. Xorshift rngs.
- [12] PHARR, M., AND HUMPHREYS, G. Physically based rendering: From theory to implementation.
- [13] WHITTET, T. An improved illumination model for shaded display. Commun. ACM 23, 6 (jun 1980), 343–349.