# BACHELOR‘S THESIS ASSIGNMENT

## I. Personal and study details

Student's name: **Maceška Ondřej** Personal ID number: **507683**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Graphics and Interaction**

Study program: **Open Informatics**

Specialisation: **Computer Games and Graphics**

## II. Bachelor’s thesis details

Bachelor’s thesis title in English:

**Direct and Indirect Illumination Computation in Unreal Engine**

Bachelor’s thesis title in Czech:

**Výpočet přímého a nepřímého osvětlení v Unreal Engine**

Guidelines:

Review existing methods for real-time direct and indirect illumination computation. Explore the Lumen technology implemented in Unreal Engine 5 and describe it in detail.
Create at least three different test scenes that demonstrate the capabilities of Lumen. The scenes will have different illumination setups and light source types, at least one scene will also contain dynamic objects. Evaluate the rendering of the created scenes regarding visual quality and rendering speed using various configurations of Lumen. Evaluate the dependency of the results on the number of light sources in the scene and the number of dynamic objects. The results will be evaluated on at least two different platforms. Based on the results, discuss the strengths and limitations of the Lumen technology.

Bibliography / sources:

[1] Daniel Wright, Krzysztof Narkowicz, and Patrick Kelly. Lumen: Real-time global illumination in Unreal Engine 5. ACM SIGGRAPH Course Notes. 2022.
[2] Tobias Ritschel, Carsten Dachsbacher, Thorsten Grosch, and Jan Kautz. The state of the art in interactive global illumination. In Computer Graphics Forum (Vol. 31, No. 1, pp. 160-188), 2012.
[3] Zander Majercik, Jean-Philippe Guertin, Derek Nowrouzezahrai, and Morgan McGuire, Dynamic Diffuse Global Illumination with Ray-Traced Irradiance Fields, Journal of Computer Graphics Techniques (JCGT), vol. 8, no. 2, 1-30, 2019
[4] Peter-Pike Sloan, Jan Kautz, and John Snyder. Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments. ACM Transactions on Graphics, Vol. 36, No. 6, Article 230, 2017.
[5] Dan Juříček. Rendering Detailed Models in Unreal Engine. Bachelor's thesis, CTU FEE, 2023.
[6] Vojtěch Vavera. Real-Time Ray Tracing in Unreal Engine. Bachelor's thesis, CTU FEE, 2020.
[7] Šimon Sedláček. Real-Time Global Illumination using Irradiance Probes. Master's thesis, CTU FEE, 2019.
[8] Antonín Šmíd. Comparison of Unity and Unreal Engine. Bachelor's thesis, CTU FEE, 2017.
[9] Anna Skorobogatova. Real-Time Global Illumination in Unreal Engine 5. Master's thesis, MU Brno, 2022.

Name and workplace of bachelor’s thesis supervisor:

**doc. Ing. Jiří Bittner, Ph.D. Department of Computer Graphics and Interaction**

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **03.09.2024** Deadline for bachelor thesis submission: **07.01.2025**

Assignment valid until: **15.02.2026**

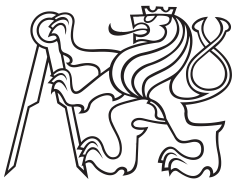
| doc. Ing. Jiří Bittner, Ph.D.<br>Supervisor’s signature | Head of department’s signature | prof. Mgr. Petr Páta, Ph.D.<br>Dean’s signature |
|---|---|---|

## III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

| . | |
|---|---|
| Date of assignment receipt | Student's signature |

Bachelor's Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

# Direct and Indirect Illumination Computation in Unreal Engine

Ondřej Maceška

Supervisor: doc. Ing. Jiří Bittner Ph.D.
January 2025

# Acknowledgements

Děkuji vedoucímu bakalářské práce doc. Ing. Jiřímu Bittnerovi Ph.D., za odborné vedení spolu s ochotným přístupem a dostatkem trpělivosti. Také jsem mu vděčný za velice podrobnou zpětnou vazbu.

Dále děkuji některým svým nejmenovaným spoužákům za důvěru, kterou měli v mou schopnost dokončit tento projekt, a za jejich dobré rady spojené s odevzdáním.


# Declaration

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Za účelem gramatické a stylistické korektury jsem využil nástroj TeXGPT s jazykovým modelem Writefull. Dále jsem použil nástroj ChatGPT pro zjednodušení tvorby grafů v kapitole 5, a to na základě mnou vytvořené šablony na předem naměřených datech. ChatGPT jsem také využil pro konzultační účely s cílem lépe porozumět některým částem dané problematiky, rychlejší vyhledávání relevantních zdrojů, a zlepšení LaTeXového formátování.

V Praze 7. ledna 2025

Podpis:.....

# Abstract

In this bachelor thesis, I analyze Lumen, a novel lighting system introduced in Unreal Engine 5, which leverages innovative techniques to compute real-time global illumination and accurate mirror reflections. I compare Lumen to other commonly employed methods for calculating direct and indirect lighting. Furthermore, I test Lumen's performance and visual quality using various scenes created in Unreal Engine 5.5.



**Keywords:** Lumen; Unreal Engine; Global illumination; Real-time rendering, Ray tracing, Radiance caching

**Supervisor:** doc. Ing. Jiří Bittner Ph.D.



# Abstrakt

V této bakalářské práci se věnuji analýze technologie Lumen. Jedná se o nový osvětlovací systém v Unreal Engine 5, který využívá inovativních konceptů pro výpočet globálního osvětlení a přesných zrcadlových odrazů v reálném čase. Porovnávám Lumen s nejvíce používanými metodami pro výpočet přímého a nepřímého osvětlení. Dále testuji rychlost a vizuální kvalitu Lumenu na různých scénách vytvořených v Unreal Engine 5.5.



**Klíčová slova:** Lumen; Unreal Engine; Globální osvětlení; Zobrazování v reálném čase; Sledování paprsků, Radiance caching

**Překlad názvu:** Výpočet přímého a nepřímého osvětlení v Unreal Engine

# Contents


**1 Introduction** 1
1.1 Goal . . . 1

**2 Rendering Fundamentals** 3
2.1 Rendering Definition . . . 3
2.2 Object Representation . . . 3
2.2.1 Polygonal Representation . . . 3
2.2.2 Volumetric Representation . . . 4
2.3 Vector Spaces in Computer Graphics . . . 5
2.3.1 Object Space . . . 5
2.3.2 World Space . . . 6
2.3.3 Camera Space . . . 6
2.3.4 Screen Space . . . 6
2.4 Lighting . . . 6
2.4.1 Light in Real World . . . 6
2.4.2 Global Illumination . . . 7
2.4.3 BRDF . . . 8
2.4.4 The Rendering Equation . . . 12
2.4.5 Monte Carlo Integration . . . 13
2.5 Light Types . . . 13
2.5.1 Hard and Soft shadows . . . 13
2.5.2 Directional Light . . . 14
2.5.3 Point Light . . . 15
2.5.4 Spotlight . . . 15
2.5.5 Area Light . . . 16
2.5.6 Ambient Light . . . 16
2.5.7 Emissive Materials . . . 16
2.6 Rendering Fundamentals Summary . . . 17

**3 Illumination Computation Techniques** 19
3.1 Light Path Expressions . . . 19
3.2 Rasterization . . . 20
3.2.1 The Rasterization Rendering Pipeline . . . 20
3.2.2 Advantages and Disadvantages 22
3.3 Ray Tracing . . . 22
3.3.1 Whitted Ray Tracing . . . 23
3.3.2 Distributed Ray Tracing . . . 24
3.3.3 Path Tracing . . . 25
3.4 Ray Tracing Optimizations . . . 25
3.4.1 Importance Sampling . . . 26
3.4.2 Bounding Volume Hierarchy . 27
3.4.3 Hardware Acceleration . . . 28
3.4.4 Screen Space Denoising . . . 29
3.4.5 Real-time Ray Tracing . . . 30
3.5 Radiosity . . . 30
3.6 Photon Mapping . . . 33
3.7 Lightmap Baking . . . 33
3.8 Illumination Techniques Summary 33

**4 Lumen Analysis** 35
4.1 Motivation . . . 35
4.1.1 The Hybrid Pipeline . . . 36
4.2 Capabilities and Limitations . . . 36
4.3 High-Level Overview . . . 37
4.3.1 Ray Tracing in Lumen . . . 39
4.4 Screen Tracing . . . 39
4.5 Software Ray Tracing . . . 41
4.5.1 Signed Distance Fields . . . 42
4.5.2 Mesh Distance Fields . . . 42
4.5.3 Sphere Tracing . . . 43
4.5.4 MDF Material Sampling . . . 44
4.5.5 Global Distance Field . . . 45
4.5.6 Height Maps . . . 45
4.5.7 Advantages . . . 46
4.5.8 Disadvantages . . . 46
4.6 Hardware Ray Tracing . . . 46
4.6.1 The Surface Cache Pipeline . 47
4.6.2 The Hit-Lighting Pipeline . . . 48
4.6.3 Far Fields . . . 49
4.6.4 Supported Hardware . . . 49
4.7 The Surface Cache . . . 50
4.7.1 Texture Atlases . . . 51
4.7.2 Cards . . . 52
4.7.3 Material Sampling . . . 57
4.7.4 Lighting Evaluation . . . 59
4.8 The Final Gather . . . 60
4.8.1 Screen Space Radiance Cache 61
4.8.2 World Space Radiance Cache 66
4.8.3 Interpolation and Integration 67
4.8.4 Temporal Filtering . . . 68
4.8.5 Other Domains . . . 69
4.9 Reflections . . . 70
4.10 Settings . . . 71
4.11 Lumen Analysis Summary . . . 72

**5 Testing Lumen** 75
5.1 Test Devices . . . 76
5.2 Feature Testing . . . 76
5.2.1 High-quality Indoor Indirect Lighting Test . . . 77
5.2.2 Hard and Soft Shadows Test 81
5.2.3 Emissive Materials Test . . . 81

5.2.4 Light Propagation Test ..... 83
5.2.5 Reflections Tests ..... 83
5.2.6 Volumetric Fog Test ..... 89
5.2.7 Feature Testing Summary ... 89
5.3 Custom Performance Testing ... 91
5.3.1 Performance Testing Scene .. 91
5.3.2 Overlap and Nanite Testing Scene ..... 91
5.3.3 Used Abbreviations ..... 93
5.3.4 Overlap and Nanite Testing Results ..... 93
5.3.5 Performance Testing Results 95
5.3.6 Performance Testing Summary ..... 105
5.4 Testing Lumen in Fortnite .... 106
5.4.1 Fortnite Introduction ..... 106
5.4.2 Fortnite Testing Description 107
5.4.3 Fortnite Setting Configurations ..... 107
5.4.4 Lumen's Impact on Fortnite's Visual Quality ..... 108
5.4.5 Lumen's Impact on Performance ..... 110

**6 Conclusion** **115**

6.1 Summary ..... 115
6.2 Lumen and the Future of Ray Tracing ..... 116
6.3 Radiance Cascades ..... 117

**A Bibliography** **119**

**B Unreal Engine Scene Controls** **125**

B.1 Feature Testing Scene Controls 125
B.2 Performance Testing Scene Controls ..... 125
B.3 Overlap Testing Scene Controls 126

**C Attached files** **127**

# Figures


1.1 An official example of a realistic indoor scene lit with Lumen in real time (source: [WN22]). . . . . 2

2.1 The Utah Teapot, one of the most famous polygonal meshes. Created by Martin Newell at the University of Utah (source: [Dun16]). . . . . 4
2.2 An example of volumetrically represented clouds (source: [Gamh]). 5
2.3 A comparison of the same objects rendered with material color only (left) and with both color and shading (right). Notice the lack of depth in the first image (source: [FS22]). . . . 7
2.4 A visualization of the wavelength spectrum visible to human eye (source: [Con24]). . . . . 7
2.5 The famous Cornell Box scene rendered only with direct lighting compared to a one-bounce global illumination. Notice the eye-pleasing color bleeding effect in the right image (source: [Jus20a]). . . . . 8
2.6 A visualization of a differential solid angle (source: [Fel23]). . . . . 9
2.7 An illustration of vectors required for calculating diffuse and specular reflections (source: [Fel23]). . . . . 11
2.8 A visualization of a general BRDF separated by its components, along with an illustration of objects with different material roughness values. The bottom three images were rendered in Blender (source: [Fel23]). . . . . 12
2.9 A visualization of hard (left) and soft (right) shadows (source: [FvBS05]). . . . . 14
2.10 A visualization of a directional light (source: [FvBS05]). . . . . 14
2.11 A visualization of a point light (source: [FvBS05]). . . . . 15
2.12 A visualization of a spotlight (source: [FvBS05]). . . . . 15
2.13 A visualization of an area light (source: [FvBS05]). . . . . 16
2.14 An example of three objects with differently colored emissive materials. . . . . 16

3.1 A simplified chart of the rendering pipeline (inspired by: [Cin23]). . . . 20
3.2 A comparison of a scene rendered using shadowless rasterization and path tracing. . . . . 23
3.3 An illustration of the Whitted ray tracing algorithm (source: [AMHH18]). . . . . 24
3.4 A comparison of two differently complex scenes rendered in Blender using the Cycles path tracer with 1, 32, and 128 samples. . . . . 25
3.5 An illustration of a bounding volume hierarchy (source: [Jus20b]). 27
3.6 An illustration of a problem with BVH (source: [Jus20b]). . . . . 28
3.7 A visualization of attributes used to calculate a form factor $F_{ji}$ (source: [AMHH18]). . . . . 31
3.8 A scene rendered using radiosity, before and after interpolation (source: [Bur09]). . . . . 32

4.1 A high level overview of Lumen's rendering pipeline. . . . . 37
4.2 Lumen's simplified ray tracing scheme. . . . . 39
4.3 An illustration of a difference between a (on the left side) and its fallback variant (on the right side) used for hardware ray tracing, along with the corresponding triangle counts (source: [WNK]). . . . . 40
4.4 An illustration of HZB traversal. In this example, the ray misses during the first step and switches to larger steps by changing the Z-buffer mip level. This occurs again after the second miss, as indicated by the increasing length of the ray's segments (source: [Lee21]). . . . . 41
4.5 A scheme of Lumen's software ray tracing pipeline. . . . . 42

4.6 An showcase of three different mip levels of a mesh SDF (source: [WNK]). . . . . 43
4.7 An illustration of the sphere tracing algorithm (source: [Tea22]). 44
4.8 A visualization of the global signed distance field (source: [WNK]). . . . 45
4.9 A scheme of Lumen's ray tracing when using hardware ray tracing. The steps highlighted in blue are performed only when using the hit-lighting pipeline. . . . . 47
4.10 A comparison of Unreal Engine 4's ray traced reflections to Lumen's hardware ray tracing pipelines (source: [WNK]). . . . . 48
4.11 A visualization of the different scene representations used in HW ray tracing. Near-field is shown in green, far-field in cyan, and the brown areas have missing surface cache coverage (source: [WNK]). . . . . 49
4.12 An example of a surface cache albedo atlas used in Epic Games' internal testing, presumably from the *Lumen in the Land of Nanite* tech demo (source: [Unr21]). . . . . 51
4.13 A visualization of a surface card (highlighted in semi-transparent green), along with its corresponding surfels (enclosed within the green bounding box), which will be projected into the surface cache during runtime. [WNK]). . . . . 52
4.14 A problem with missing surface cache coverage. Parts of the mesh without corresponding cards are highlighted in pink. Back faces are shown in black (source: [Gamc]). . 53
4.15 A scheme of Lumen's card generation process (source: [WNK]). 53
4.16 A visualization of the cluster regrowing process and an object fully covered in surfel clusters (source: [WNK]). . . . . 55
4.17 A comparison of a rendered scene and its visualized Nanite representation with two different levels of detail (source: [Ver21]) . . . 55
4.18 A comparison between a building covered by separate cards and its variant where all cards are merged into one (source: [WNK]). . . . . 56
4.19 A visualization of a simple mesh covered with surfel cards. Any surface point inside the corner triangular face is sampled using all three cards, as it lies within all their bounding boxes. This is indicated by the fact that all components of its squared normal vector have positive values. To retrieve correct cached material and lighting colors, the results of all three cards need to be weighted and blended together to avoid a stretched perspective. . . . . 58
4.20 A scheme showing Lumen's surface cache lighting update methods. . . . . 59
4.21 A scheme showing Lumen's final gather. Note that all its steps except for radiance caching are performed at full resolution. . . . . 61
4.22 A visualization of the screen space radiance cache probe atlas. Note that all adaptively placed probes are stored at the bottom of the atlas to avoid the necessity of using another data structure and separate processing step (source: [Wri21]). . 62
4.23 A visualization of Lumen's hierarchical refinement for probe placement, where every white dot represents a single probe from the screen space radiance cache. At each level, the pixels with failed interpolation are highlighted in red (source: [Wri21]). . . . . 63

4.24 A visualization of structured product importance sampling in the texture domain. Most rays stored at the top are culled due to low BRDF and some of the rays with large BRDF and lighting product are super-sampled (source: [Wri21]). . . 64
4.25 A comparison of a scene rendered using Lumen with uniform (left) and structured product importance sampling (right). The rays originating from one screen probe are visualized, with the super-sampled ones highlighted in white. Notice the substantially lower amount of noise in the right image, especially in the corners of the room (source: [WNK]). . . . . 65
4.26 A comparison of an image using an unfiltered and filtered radiance cache. Notice the substantially lower amount of noise in the right image (source: [Wri21]). . . . . 66
4.27 A visualization of the world space radiance cache used for distant lighting. Notice the sparse allocation around visible objects. . . . . 67
4.28 An example of a scene rendered using Lumen's final gather, along with its screen space radiance cache and full-resolution rasterized normals stored in a buffer (source: [Wri21]). 68
4.29 A showcase of Lumen's volumetric final gather and its multiple steps (source: [WNK]). . . . . 69
4.30 A showcase of probe placement in Lumen's final gather used for propagating indirect lighting within the surface cache (source: [WNK]). 70
4.31 An example of a scene rendered with both Lumen GI and reflections (source: [Gam23]). . . . . 71
5.1 An example of an indoor scene lit primarily by indirect lighting. The skinned character properly receives indirect lighting due to screen traces, despite having no MDF representation and surface cache coverage. This image was rendered using Lumen Default and software ray tracing on PC 1. . . . . 78
5.2 A subtle color bleeding effect caused by both objects can be seen on the ceiling, floor, and walls. This image was rendered using Lumen Default and hardware ray tracing on PC 1. . . . . 79
5.3 Apparent "Light leaking" seen in an unlit room. This image was rendered using Lumen Default and software ray tracing on PC 1 in Unreal Engine 5.3. . . . . 80
5.4 A strange blue spot artifact appearing in a closed, unlit room. This image was rendered using Lumen High and hardware ray tracing on PC 1. . . . . 80
5.5 A showcase of three types of light sources correctly casting shadows with varying softness. This image was rendered using hardware ray tracing on PC 1 in Unreal Engine 5.3. . . . . 81
5.6 A room correctly lit by a visible emissive object. This image was rendered using Lumen Default and software ray tracing on PC 1. . . . . 82
5.7 A room incorrectly lit by a disabled emissive object. This image was rendered using Lumen Default and hardware ray tracing on PC 1. 83
5.8 An apparent mismatch between a color of a diffuse object and its reflection in its neighboring reflective object. This image was rendered using hardware ray tracing on PC 1 in Unreal Engine 5.3. . . . . 84

5.9 A showcase of indoor mirror reflections. Notice the low-frequency noise, almost entirely missing character, completely broken texture of the emissive object and the rough distance field representation of the blue object. This image was rendered using Lumen Default and software ray tracing on PC 1. . . . . 85
5.10 A showcase of indoor mirror reflections. Notice the almost entirely black character and other visual artifacts, such as the supposed partially missing surface cache coverage of the blue diffuse object. This image was rendered using Lumen High and hardware ray tracing on PC 1. . . . . 86
5.11 A comparison of outdoor glossy reflections with varying roughness rendered using Lumen Default and Lumen High with software ray tracing on PC 1. Notice the terrain missing in the reflection when using Lumen Default. . . . . 87
5.12 A comparison of multi-bounce mirror reflections rendered using Lumen High on software and hardware ray tracing on PC 1. Notice the black character and downsampled skybox in reflections. 88
5.13 A showcase of many view-dependent artifacts caused by Lumen's Reflections. Notice the partially missing secondary bounce inside the mirror on left. The terrain is also missing in all reflections, except for the part where screen space tracing has usable data to work it. Part of the mirror reflection next to the character also appears incorrectly. This image was rendered using Lumen Default and software ray tracing on PC 1. . . . . 88
5.14 A massive light leaking issue caused by a local volumetric height fog with scattering distribution of 0.8. This image was rendered using software ray tracing in Unreal Engine 5.3. . . . . 89
5.15 A showcase of the Performance Testing scene containing 100 000 objects, 255 point lights and no directional light. . . . . 92
5.16 A showcase of the Overlap Testing scene containing 10 000 objects. . . 92
5.17 Overlap and Nanite test results with 8 various settings, measured on PC 1. . . . . 94
5.18 Overlap and Nanite test results with 8 various settings, measured on PC 3. . . . . 95
5.19 A comparison of results for 50, 100, and 255 lights with ML on and off, measured on PC 1 with HWRT, Lumen Default, and opaque walls. 97
5.20 A comparison of results for 50, 100, and 255 lights with ML on and off, measured on PC 1 with HWRT, Lumen High and opaque walls. . . . 98
5.21 A comparison of software and hardware ray tracing results for 0, 50, and 255 lights with ML off, measured on PC 1 with Lumen Default and opaque walls. . . . . 99
5.22 A comparison of software and hardware ray tracing results for 0, 50, and 255 lights with ML on, measured on PC 3 with Lumen Default and opaque walls. . . . . 100
5.23 A comparison of Lumen Default and Lumen High results for 0, 50, and 255 lights with ML on, measured on PC 3 with hardware ray tracing and opaque walls. . . . . 101
5.24 A comparison of results with opaque (O) and reflective (R) walls, using 0, 50, and 255 lights with ML off, measured on PC 3 with Lumen Default and software ray tracing. 102

5.25 A comparison of results with opaque (O) and reflective (R) walls, using 0, 50, and 255 lights with ML on, measured on PC 3 with Lumen High and hardware ray tracing. . . 103
5.26 A comparison of results obtained from testing the influence of dynamic objects using Setting 1 and Setting 2 on PC 1 and PC 3. . . . . 104
5.27 A screenshot of the Performance Testing scene with 18 000 objects. Notice how the majority of objects does not appear in the mirror reflection on left. Furthermore, some of those which do appear have missing surface cache coverage, thus partially showing as black. This image was rendered using hardware ray tracing with Lumen High, Nanite and the Hit-Lighting pipeline. . . . 105
5.28 A visual comparison between hardware and software ray tracing. Rendered with Nanite and Lumen Default. . . . . 105
5.29 A visual comparison of Fortnite's visuals when using no global illumination and with Lumen enabled. Rendered on PC 2. . . . . 109
5.30 A visual comparison of Fortnite's outdoor scenery when using software (SWRT) and hardware ray tracing (HWRT). Notice the lack of direct shadowing on the player character and the missing storm when using SWRT. Rendered on PC 2 using the Lumen Epic configuration. . . . . 110
5.31 A showcase of a visual artifact caused by Lumen's reliance on screen space ray tracing in Fortnite. Notice the partial lack of building in the reflection on the water surface. The character's gun was highlighted in red for better visual clarity, but no further modifications were made to the image. Rendered on PC 2 using the Lumen Epic configuration with disabled DLSS. . . . . 111
5.32 A comparison of Fortnite's performance using 5 different setting configurations. The x-axis is limited only to the interval from 60 to 180 seconds to improve visual clarity. This Figure was created using CapFrameX and all data was measured in January 2025 on PC 2. 112
5.33 A comparison of Fortnite's average and median performance using 5 different setting configurations. This Figure was created using CapFrameX and all data was measured in January 2025 on PC 2. . . . . 112
5.34 An older comparison of Fortnite's performance using 4 different setting configurations. The x-axis is limited only to the interval from 60 to 180 seconds to improve visual clarity. This Figure was created using CapFrameX and all data was measured in May 2024 on PC 2. . 113
5.35 An older comparison of Fortnite's average and median performance using 4 different setting configurations. This Figure was created using CapFrameX and all data was measured in May 2024 on PC 2. . . . . 113

6.1 A comparison of Path of Exile 2's visual quality with no global illumination and with radiance cascades GI. . . . . 117

# Tables

5.1 Hardware specifications of all three devices I used for testing. . . . . 76
5.2 The various Fortnite setting configurations I tested. Note that **SR** stands for super-resolution and **LL** for low-latency. . . . . 107

# Chapter 1

# Introduction

With the ever-growing public interest in game engines and their increasing usage across various industries even beyond gaming, it comes as no surprise that companies are incentivized to innovate and push their technology forward. A perfect example of such recent innovation is Epic Games' *Unreal Engine 5*, a major update to an already well-established framework that aims to simplify the workflow of digital artists who use the engine as their primary tool. It achieves this by introducing two major overhauls to its rendering systems [Gamf, Fla24].

The first of these upgrades is a virtualized geometry solution called *Nanite*. This technology uses dynamic triangle clustering and streaming to render only the visible parts of each 3D mesh, thus eliminating the need to manually and laboriously create its level-of-detail variations. This more efficient representation of geometry also leads to a significant decrease in the total number of GPU draw calls, which dramatically increases the performance when rendering enormous scenes with highly detailed geometry [Ju3].

The second major new system in Unreal Engine 5 is *Lumen*, a hybrid rendering pipeline that enriches Unreal Engine scenes with real-time global illumination and mirror reflections, as shown in Figure 1.1. Using indirect lighting bounces to calculate global illumination in real time is one of the hardest problems in computer graphics. Lumen supposedly achieves this while also being able to run on devices without graphics processing units specialized for ray tracing [Gam23].

## 1.1 Goal

In this thesis, I will analyze and describe some of the most commonly used approaches to calculate both direct and indirect lighting and contextually relate them to Lumen. Afterwards, I will identify and describe the ingenious methods and optimization tricks Lumen uses to achieve its seemingly photo-realistic results in real time, as well as discuss Nanite's importance during this process. Furthermore, I am going to measure Lumen's performance and point out some of the visual artifacts that can occur when using it. This testing will be performed not only on custom scenes created in Unreal Engine 5, but also on one of the world's most popular video-games which supports

**Figure 1.1:** An official example of a realistic indoor scene lit with Lumen in real time (source: [WN22]).

both Nanite and Lumen and comes directly from Epic Games - *Fortnite*.

# Chapter 2

# Rendering Fundamentals

Before addressing the various methods used to calculate realistic lighting, it is important to define the necessary terminology and explain the most crucial high-level rendering concepts. Feel free to skip this entire Chapter if you are already familiar with the basics of computer graphics.

## 2.1 Rendering Definition

*Rendering* is commonly referred to as the process of generating a two-dimensional image or a series of images from a three-dimensional scene. This process includes, but is not limited to [FvBS05, AMHH18]:

- efficiently representing objects and their material properties,
- simulating a virtual camera and projecting the 3D scene on a 2D plane from its point of view, then mapping this plane to our screen,
- shading, lighting, and shadowing each object by calculating direct and indirect illumination,
- applying image post-processing effects, such as anti-aliasing, bloom, depth of field or motion blur.

## 2.2 Object Representation

In this Section, I will introduce the most common approaches used to represent a 3D object in computer memory, as knowing the differences between them will be crucial to later understand Lumen's custom software ray-tracing pipeline (described in Section 4.5) [FvBS05].

### 2.2.1 Polygonal Representation

The most commonly used technique for representing 3D geometry is called *boundary representation* (often abbreviated as B-Rep), where, as the name suggests, each object is defined strictly by its boundary. However, this boundary can sometimes be a very complex shape, so we typically approximate

it using simple polygons, such as triangles, which are easy to store in memory. They can also be efficiently rendered using a graphic processing unit (GPU).

Each triangle is composed of three vertices located in a 3D space, along with their normal vectors and additional optional data, such as vertex colors or UV coordinates, which are used for texturing.

Together, these triangles form what is defined as *polygonal mesh*. When modeling, we also need information about the topology of this mesh, i.e. how the neighboring vertices are connected by edges. However, for the purposes of rendering, we do not require this additional information, and the aforementioned attributes are usually sufficient. This subcategory of B-Rep is called *polygonal representation* and is primarily used in video games and movies. It is not well suited for industrial modeling, where precise mathematical descriptions are required [FvBS05, AMHH18].

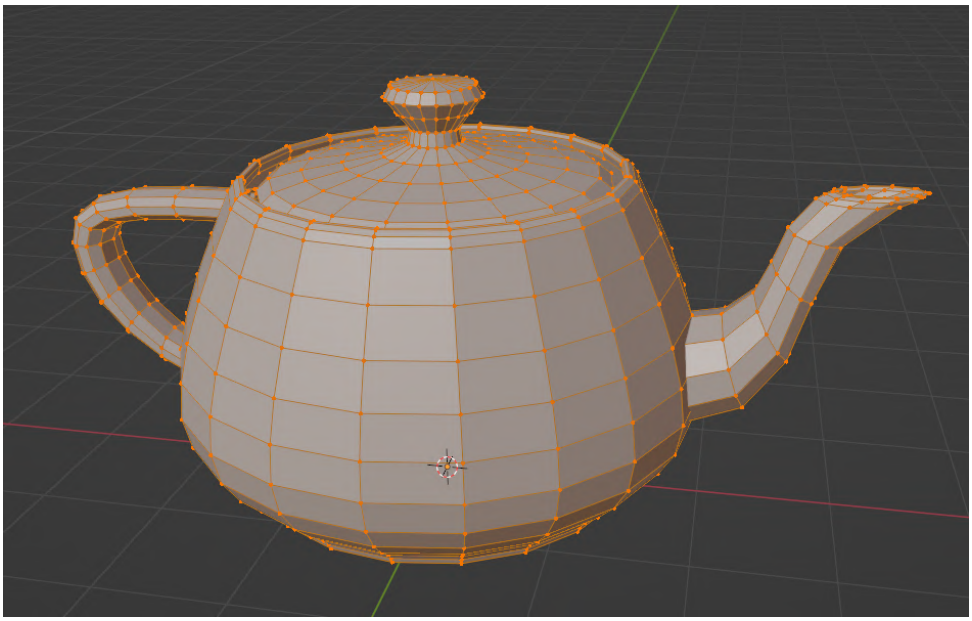


**Figure 2.1:** The Utah Teapot, one of the most famous polygonal meshes. Created by Martin Newell at the University of Utah (source: [Dun16]).

### 2.2.2 Volumetric Representation

While polygonal representation can be used to approximate most real-world objects, there are some exceptions, such as clouds, fog, or smoke. In these cases, a more suitable representation is needed to convey a sense of depth. This is also true for situations where the internal structure of an object is equally as important as its boundary.

This more suitable representation is called *volumetric representation*. We typically use *voxels* (volumetric pixels) to store such objects in computer memory. Voxels can be thought of as small cubes placed on a uniform 3D grid, similar to pixels on our screen but with an additional spatial coordinate.

I describe one well-known method used to render volumetric objects, called *sphere tracing*, in Section 4.5.3 [FvBS05, AMHH18].

**Figure 2.2:** An example of volumetrically represented clouds (source: [Gamh]).

## 2.3 Vector Spaces in Computer Graphics

In computer graphics, various vector spaces and coordinate systems are used for different parts of the rendering process. Similarly, some lighting techniques operate solely within the confines of a single vector space. In this Section, I will briefly introduce a few of the most important spaces and describe their differences and use cases. Transformations between these coordinate spaces are mentioned but not explained in detail, as they are not particularly relevant for understanding the concepts of calculating lighting. If you wish to learn more about vector spaces and the linear algebra behind them, please refer to Felkel [FvBS05] or Akenine-Möller et al.[AMHH18].

### 2.3.1 Object Space

As already mentioned, polygonal surfaces are usually represented by triangles, where each triangle is composed of three vertices. These vertices are essentially points in an orthonormal three-dimensional Cartesian vector space often referred to as *object space*, local space, or modeling space. Each mesh has its own local space, and the vertices of which it is made are typically centered around the origin of this space.

This is the coordinate system an artist typically operates in when creating a mesh for a 3D model in dedicated software, such as *Blender* or *Autodesk Maya*. However, object space is generally not used for lighting calculations.

### 2.3.2 World Space

The shared coordinate system of a scene where all objects are individually placed, rotated, and scaled is called *world space*. Transitioning to this space from object space is achieved by transforming all the vertices of each object by a multiplication with a model matrix, also known as the transformation matrix.

### 2.3.3 Camera Space

*Camera space* (also called view space or eye space) is a world space transformed by a view matrix. This transformation translates all objects so that the camera's position is shifted to the origin, and two orthogonal vectors are clearly defined, one pointing upward and the other to the right of the camera. These vectors, typically called the *up* and *right* vectors, allow us to calculate the camera's view direction, which is necessary to know for correct projection to *screen space*.

### 2.3.4 Screen Space

The integer range of the x and y coordinates in this two-dimensional space directly corresponds to pixels on the part of the screen to which we render.

## 2.4 Lighting

Calculating light's interaction with surfaces, also known as shading, is arguably the most important part of rendering. Without proper lighting contribution, the resulting image may appear to lack depth, clarity, and detail, as shown in Figure 2.3. Similarly, without shadows, it can be difficult to correctly determine object's distance from camera or its spatial relation to other objects in the scene [FS22].

Other desirable photorealistic effects, such as mirror reflections (often simply called *reflections*), caustics, and color bleeding, are very difficult, if not impossible, to simulate without an appropriate approximation of physically based light reflection calculations [KG09, Kim22b].

### 2.4.1 Light in Real World

In physics, monochromatic light is described as electromagnetic radiation with a single *wavelength* denoted as $\lambda$. The light visible to the human eye corresponds to wavelengths ranging from 380 to 720 nanometers (Figure 2.4), where the shorter wavelengths are perceived as blue and the longer wavelengths as red, with green in between [FvBS05].

Using geometric optics, light in general is then typically defined as a combination of multiple monochromatic light rays with different wavelengths. After each ray is emitted from a light source, it eventually intersects with a surface. During this collision, part of the ray's wavelength range is absorbed

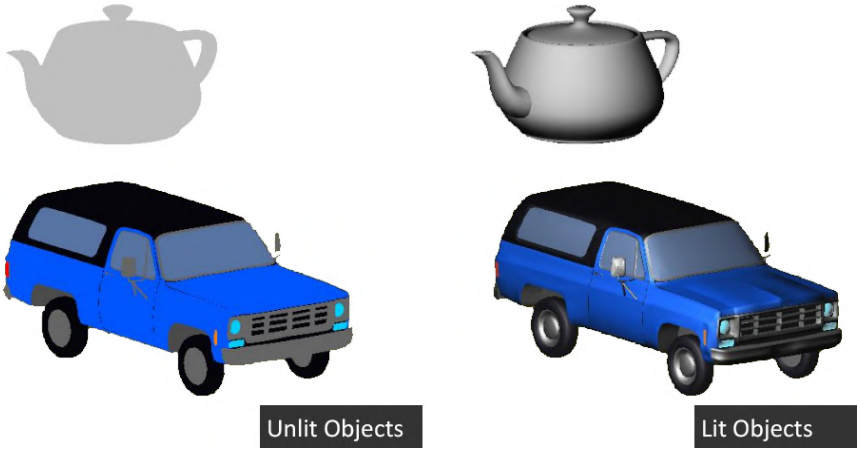



**Figure 2.3:** A comparison of the same objects rendered with material color only (left) and with both color and shading (right). Notice the lack of depth in the first image (source: [FS22]).

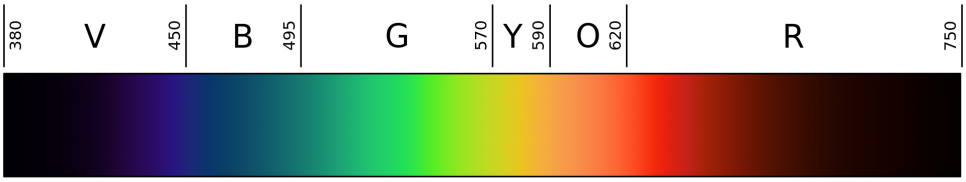



**Figure 2.4:** A visualization of the wavelength spectrum visible to human eye (source: [Con24]).

based on the material properties, while other wavelengths are reflected, effectively "tinting" the ray [FvBS05, FS22].

In computer graphics, especially when rendering photorealistic scenes, these physical principles are mostly adhered to, with a few exceptions. First, the wavelength value of each light ray is approximated by three numbers corresponding to the red, green, and blue values in a selected additive color model. This vector is commonly known as *light intensity*. If you wish to learn more about related color theory, please refer to Gravesen [Gra15] or Tychtl [Tyc24].

Second, light rays are generally traced from camera to the light sources, rather than the other way around, as calculating rays that never reach the viewer would be a waste of computational resources. This approach is explored more thoroughly in Section 3.3 when discussing ray tracing [FS22, Fel23, KG09].

### 2.4.2 Global Illumination

When a light ray traced from camera hits a point on a surface, we can reflect it towards a light source and calculate the incoming direct lighting and that point. This is typically very fast to compute when using methods like *rasterization*, as to shade each object, we only need to know its direction

and/or distance to each light source, along with its material properties.

However, to achieve physically-based rendering, we also need to calculate shadows and the incoming indirect lighting reaching that point, i.e. the rays which are reflected to this point from other nearby surfaces. This phenomenon is called *interreflection* and it results in a visually appealing optical effect called *color bleeding*, which causes surfaces to be slightly tinted by their neighboring objects, as shown in Figure 2.5. Indirect lighting also ensures proper visibility of objects that do not receive any direct light, as their surface would otherwise appear completely black. This problem is usually mitigated by relying on omnidirectional ambient lighting, which is described in Section 2.5.6) [KG09].

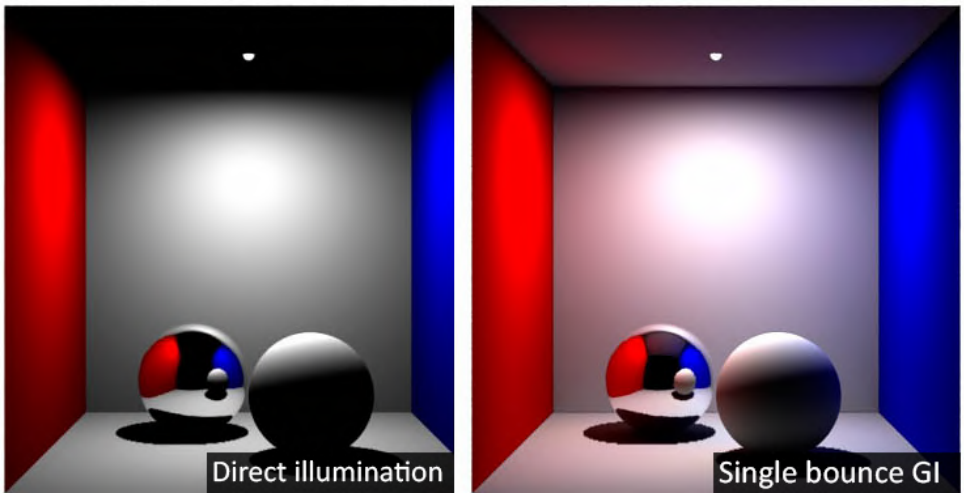



**Figure 2.5:** The famous Cornell Box scene rendered only with direct lighting compared to a one-bounce global illumination. Notice the eye-pleasing color bleeding effect in the right image (source: [Jus20a]).

*Global Illumination* refers to the calculation of both direct and indirect lighting, which is one of the hardest problems in computer graphics. There are many different algorithms that can compute it, all of which can be described as solving a variation of a famous equation known as *the rendering equation* (described in Section 2.4.4). However, to fully understand this equation, we first need to introduce one of the most important terms in rendering, *BRDF* [Kim22b, KG09, AMHH18].

### 2.4.3 BRDF

The *Bidirectional Reflectance Distribution Function* (BRDF) is a function of four real variables that, given a normalized incoming direction $\omega_\mathrm{i}$ and a normalized outgoing direction $\omega_\mathrm{o}$, returns the ratio of *radiance* $L_\mathrm{o}(\omega_\mathrm{o})$ reflected along the direction $\omega_\mathrm{o}$ to the *irradiance* $E_\mathrm{i}(\omega_\mathrm{i})$ at a surface point $x$. The units of BRDF are inverse steradians [Fel23, KG09].

$$BRDF(\omega_\mathrm{i}, \omega_\mathrm{o}) = \frac{\mathrm{d}L_\mathrm{o}(\omega_\mathrm{o})}{\mathrm{d}E_\mathrm{i}(\omega_\mathrm{i})} = \frac{\mathrm{d}L_\mathrm{o}(\omega_\mathrm{o})}{L_\mathrm{i}(\omega_\mathrm{i}) \cos\theta_\mathrm{i} \mathrm{d}\omega_\mathrm{i}} \tag{2.1}$$

Below is the explanation of the individual components of BRDF along with a clarification of the coordinate system used to represent the unit vector directions.

### Spherical Coordinates

The incoming and outgoing light directions tend to be represented as two spherical angles rather than as three-dimensional Cartesian vectors described by their $x$, $y$, and $z$ coordinates.

$$\omega = (\omega_x, \omega_y, \omega_z) = (\theta, \varphi) \tag{2.2}$$
$$\theta \in \langle 0, \pi \rangle \tag{2.3}$$
$$\varphi \in \langle 0, 2\pi \rangle \tag{2.4}$$

$\varphi$ is the azimuth angle around the surface normal, whereas $\theta$ is referred to as polar angle. Following equations describe the transition from spherical to Cartesian coordinate representation:

$$\omega_x = \sin\theta \cos\varphi \tag{2.5}$$
$$\omega_y = \sin\theta \sin\varphi \tag{2.6}$$
$$\omega_z = \cos\varphi \tag{2.7}$$

And the other way around:

$$\theta = \arccos\omega_z \tag{2.8}$$
$$\varphi = \arctan\frac{\omega_y}{\omega_x} \tag{2.9}$$

The differential solid angle (visualized in Figure 2.6), denoted $d\omega$, is a small flat surface patch on a unit sphere.

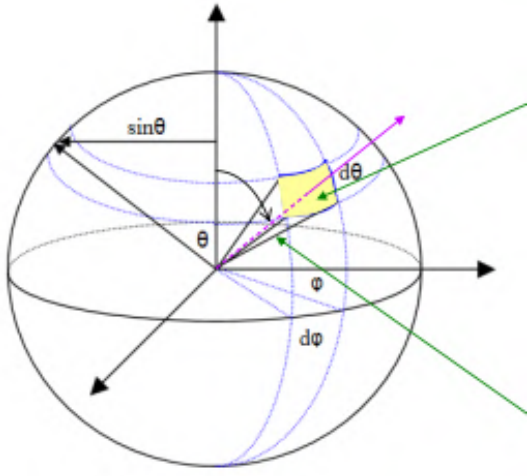



**Figure 2.6:** A visualization of a differential solid angle (source: [Fel23]).

### Radiance

Radiance $L_o(\omega_o)$ stands for radiant flux, which is the amount of light emitted and reflected in direction $\omega o$ from a surface point $x$. It is measured in unit power per solid angle per unit projected area. To approximate the behavior of light traveling through air, radiance is considered to be constant along a ray. This means that the incoming radiance $L_i(\omega_i)$ at the point $x$ is equal to the outgoing radiance $L'_o(\omega'_o)$ from a visible point $x'$. This simplification is very important for understanding ray tracing (described in Section 3.3) and path tracing (described in Section 3.3.3) [Fel23, KG09, FvBS05].

### Irradiance

Irradiance is the total radiant flux received by a surface point $x$. It is measured in units of power per unit projected area [FS22, FvBS05].

### Properties of BRDF

To achieve physically plausible results, the BRDF must satisfy the following properties: [FvBS05]

- Positivity: $BRDF(\omega_i, \omega_o) \geq 0$
- Reciprocity: $BRDF(\omega_i, \omega_o) = BRDF(\omega_o, \omega_i)$
- Conservation of energy: $\forall \omega_i, \int_\Omega BRDF(\omega_i, \omega_o) \cos\theta_i \, d\omega_i \leq 1$
- Linearity: The value of $BRDF$ for any incoming direction $\omega_i$ does not depend on the values of $BRDF$ for any other incoming directions.

### Reflection Types

In practice, BRDF is used to model the reflective properties of a surface. Below are descriptions of the two main reflection models used in computer graphics. Their contribution to a general BRDF is visualized in Figure 2.8 along with three material examples rendered in Blender using path tracing. All vectors used in the following equations are shown in Figure 2.7.

- *Diffuse Reflection* occurs when a light ray hits an object with many surface-level microscopic imperfections, such as wood or dry paint. The reflected light is then equally scattered in all directions, resulting in its perceived intensity being independent of the view direction. The *Lambertian reflectance* model is typically used to approximate diffuse reflection:

$$I_D = I_L r_D max((\vec{l} \cdot \vec{n}), 0) \tag{2.10}$$

  Here, $I_D$ stands for the reflected intensity, $I_L$ denotes the intensity of the light source, and the vector $r_D$ is the diffuse coefficient (we can think

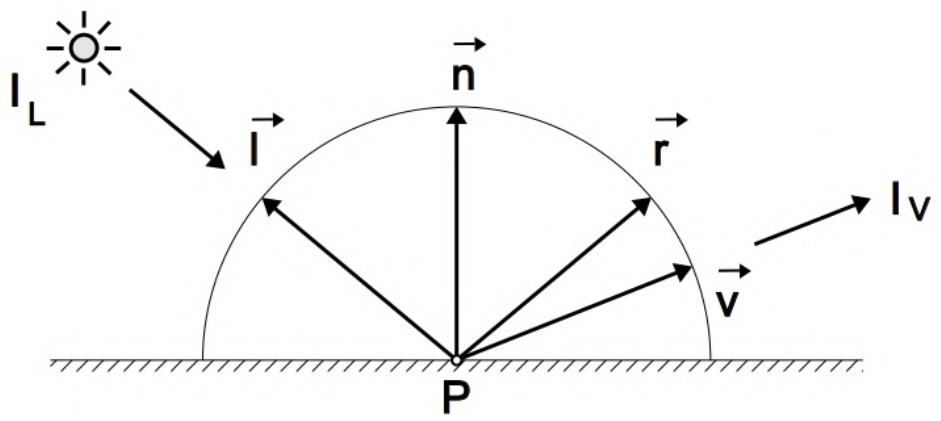



**Figure 2.7:** An illustration of vectors required for calculating diffuse and specular reflections (source: [Fel23]).

of it as the color of the surface represented by RGB values). The dot product of the surface normal $\vec{n}$ and the direction to light $\vec{l}$ determines the fraction of the reflected intensity. This corresponds to the cosine of the angle between these normalized vectors and must be clamped to 0 to eliminate the negative contribution of the light source in case it comes from behind the surface [FvBS05, AMHH18].

- *Specular Reflection* occurs when a light ray hits an object with a perfectly smooth surface, such as a mirror, and is reflected in a single specific direction. The intensity of the reflection strongly depends on the angle between the direction of the ideal reflection $\vec{r}$ and the direction to camera $\vec{v}$. The *Phong* model (whose specular part is shown below) or the slightly faster *Blinn-Phong* model is typically used to approximate specular reflections.

$$I_S = I_L r_S max((\vec{v} \cdot \vec{r})^h, 0) \tag{2.11}$$

  Here, the scalar coefficient $h$ stands for shininess, and can range from 0 to $\infty$ [FvBS05, AMHH18].

- *Glossy Reflection* is used for materials that are neither perfectly rough nor perfectly smooth. To control how much this reflection resembles a diffuse or specular reflection, the glossiness parameter (often referred to as roughness) is used. With lower values, the distribution of sampled directions approaches the single-ray approximation of a sharp specular reflection. With higher values, the distribution becomes wider, resulting in a blurrier specular reflection. This distribution is called *the reflectance lobe*.

For more information on reflection models, specific mathematical formulas, and their use cases, please refer to Akenine-Möller et al. [AMHH18]. A

common notation used to describe a light path with any combination of these reflections is shown in Section 3.1.

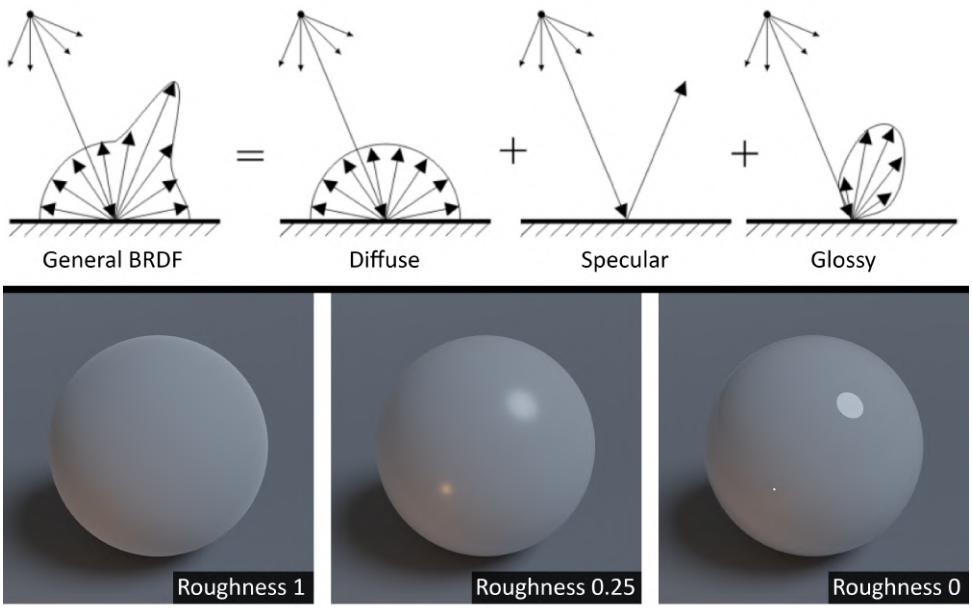



**Figure 2.8:** A visualization of a general BRDF separated by its components, along with an illustration of objects with different material roughness values. The bottom three images were rendered in Blender (source: [Fel23]).

One limitation of BRDF is that it only applies to opaque surfaces. To simulate semi-transparent materials such as glass or ice, calculating *transmission* of light is required. This is often done by using a *BTDF* (Bi-Directional Transmittance distribution function), which is then combined with BRDF. Note that BTFD extends the computational domain to a whole sphere [RDGK12]. *Subsurface scattering* is another effect that cannot be modeled using a basic BRDF, as it requires information about the object's interior.

### 2.4.4 The Rendering Equation

The aforementioned rendering equation, founded by James Kajiya in 1986 [Kaj86], reads as follows:

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega L_i(x, \omega_i) BRDF_r(x, \omega_o, \omega_i) \cos\theta_i \, d\omega_i \qquad (2.12)$$

This famous equation describes how the outgoing radiance $L_o$ at any point $x$ in direction $\omega_o$ is equal to the light emitted from that point $L_e$ (in case it belongs to a light source or an object with emissive material) plus the sum of reflected radiance coming from all directions $\omega$ within a normalized hemisphere $\Omega$ centered around the normal vector of the surface.

To calculate global illumination, we need to solve this equation for every pixel on the screen. This is a recursive problem, as the incoming radiance

$L_i(x, \omega_i)$ at point $x$ is equal to the outgoing radiance $L_o(y, \omega_i)$ at some other point $y$ [Jus20a, Wri21, KG09, AMHH18].

### 2.4.5 Monte Carlo Integration

As integrating over a continuous hemisphere is generally not computationally feasible, a numerical approach is often used to approximate the solution to the rendering equation: *Monte Carlo integration* [Jus20a, Wri21, KG09]

Imagine any given multidimensional definite integral, with a domain $\Omega \subset R^m$.

$$I = \int_{\Omega} f(\mathbf{x})\, d\mathbf{x} \tag{2.13}$$

Instead of relying on analytical computation, this method gets the estimated result by sampling the function for many random input points inside the integration domain and averaging the results. This works because the *law of large numbers* states that with an increasing number of independent random samples, their arithmetic mean multiplied by the volume of the sampling domain converges to the true value [Sed15, Wri21, Jus20a].

$$\int_{\Omega} f(\mathbf{x})\, d\mathbf{x} \approx \lim_{N \to \infty} \frac{Vol(\Omega)}{N} \sum_{i=1}^{N} f(x_i) \tag{2.14}$$

In the most basic form of this algorithm, all samples are uniformly distributed. However, this is not always ideal, as many of the samples may contribute little to no value to the final result. For example, a single ray cast from a surface in one direction can completely light coming from another direction.

Therefore, a better sampling approach is discussed in Section 3.4.1 and a more specific version of Monte Carlo integration used in lighting calculation is presented in Sections 3.3.2 and 3.3.3 [KG09].

## 2.5 Light Types

With reasons for achieving global illumination and the basic concepts behind calculating it covered, I find it important to describe the different types of light sources, how we represent them in a 3D scene, and what are their typical use cases [FS22, FvBS05, RDGK12, AMHH18].

### 2.5.1 Hard and Soft shadows

Understanding the distinction between *hard* and *soft* shadows (shown in Figure 2.9) is essential to grasp the differences between the different types of light sources.

Imagine a scene with an infinitely small light source and a single object, called *receiver*. If there is another object *(occluder)* between a point on the

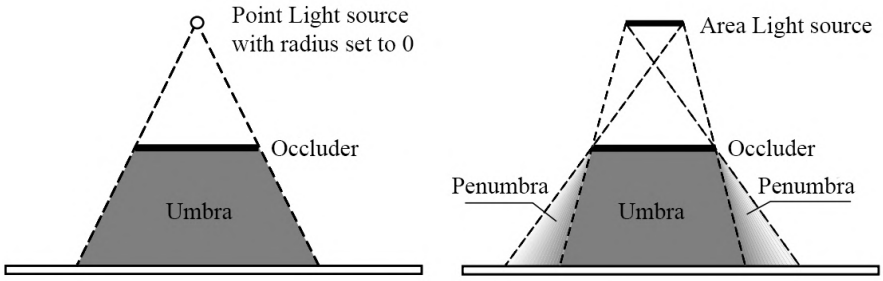



**Figure 2.9:** A visualization of hard (left) and soft (right) shadows (source: [FvBS05]).

receiver's surface and the light source, that point is in the occluder's shadow. Otherwise, it is lit by that light source, with no options in between.

However, real light sources have a non-zero surface area and, as such, they tend to cast shadows that appear as having blurred (soft) edges. This is because some shadowed points on the receiver are also lit by light rays coming from the same light source, as illustrated in Figure 2.9. This blurred part of the shadow is called *penumbra*, while the part that no light rays reach is referred to as *umbra*. Increasing the surface area of the light source results in larger penumbras and smaller umbras, and vice versa.

This implies that shadows with sharp edges, i.e. those with no penumbra, are cast only by infinitely small light sources or light sources that are infinitely far away from the lit objects. We refer to these as hard shadows [FvBS05, AMHH18].

### 2.5.2 Directional Light

Directional light is the most commonly used type of light, found in nearly all outdoor scenes. It is a point infinitely far away from all rendered objects, and as such, we typically represent it using only a normalized direction vector and color attributes.

Directional lights have no *attenuation*, which means that their intensity is not influenced by their distance to the objects they illuminate. This makes them ideal for simulating the Sun or other distant and powerful light sources that produce seemingly hard shadows [FvBS05].

**Figure 2.10:** A visualization of a directional light (source: [FvBS05]).

### 2.5.3 Point Light

Point lights are placed directly in the world space and emit equally strong light in all directions. In addition, they may have attenuation, typically composed of three factors: constant, linear, and quadratic. This is shown in the following equation, where each $k$ stands for an arbitrary constant value and $d$ is the distance of the illuminated surface from the point light [FvBS05].

$$Attenuation = \frac{1}{k_C + k_L * d + k_Q * d^2} \tag{2.15}$$

Note that in the real world, light follows *the inverse square law*, which implies that only quadratic attenuation can produce seemingly realistic results. In contrast, constant and linear attenuation values allow for more artistic freedom that is not necessarily based on real-world physics [GR21, FvBS05].

Point lights are typically used to simulate indoor light sources, such as light bulbs, candles, or torches. However, as previously mentioned, these objects are neither infinitely small nor infinitely far away. To give point lights the ability to cast soft shadows, an additional parameter is often used, which modifies the light's radius [FvBS05].

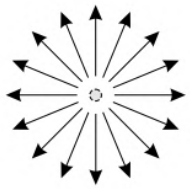


**Figure 2.11:** A visualization of a point light (source: [FvBS05]).

### 2.5.4 Spotlight

Spotlights are quite similar to point lights, albeit with one key difference. Instead of representing an omnidirectional light source, spotlights focus the emitted light in a cone with specific direction and radius.

Additional parameters, such as the spot cutoff and exponent, are often used to further modify the exact distribution of light within its cone.

Spot lights are often used to represent flashlights or car lights [FvBS05].

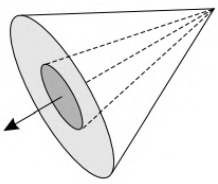


**Figure 2.12:** A visualization of a spotlight (source: [FvBS05]).

### 2.5.5 Area Light

This type casts light from a single rectangular face with an arbitrary area. It produces soft shadows and is typically used in indoor scenes. Sometimes, it is referred to as rectangular light.

Area lights can be used to approximate light coming from the ceiling or softbox lights used in professional photography [FvBS05].

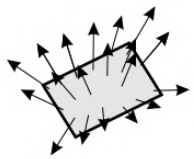


**Figure 2.13:** A visualization of an area light (source: [FvBS05]).

### 2.5.6 Ambient Light

Ambient lighting is used to very roughly approximate indirect lighting in scenes where only direct lighting bounces are calculated. Without diffuse inter-reflections, unlit parts of the scene would otherwise appear completely black. Ambient light is typically applied uniformly to the surface of every object in the scene regardless of its transformation and is generally not used in physically-based rendering approaches [FvBS05, RDGK12].

### 2.5.7 Emissive Materials

Materials of objects in the scene can also have emissive properties, as shown in Figure 2.14.

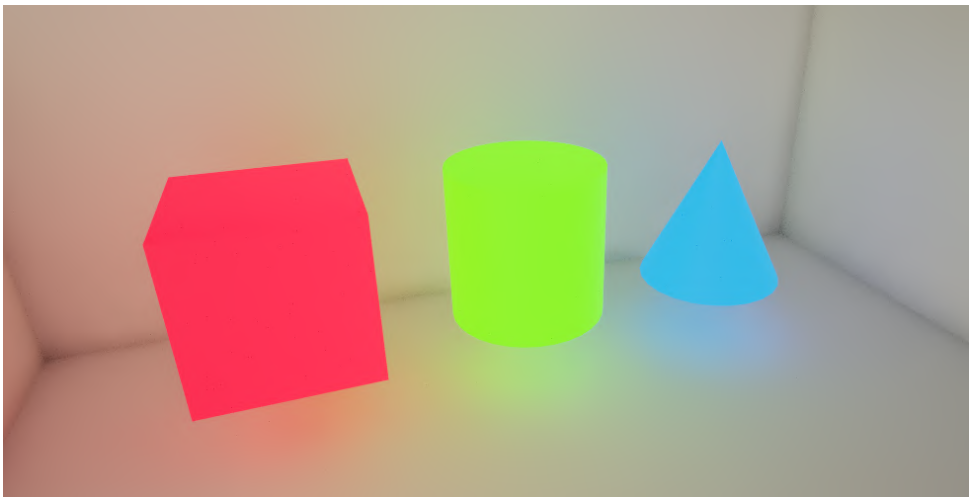


**Figure 2.14:** An example of three objects with differently colored emissive materials.

Note that emissive objects will typically illuminate their surroundings only when indirect lighting bounces are calculated, as they are not considered to be actual light sources. In strictly direct lighting methods such as rasterization (Section 3.2), a screen-space post-process effect called *bloom* can be used to naively approximate the illumination coming from emissive surfaces [AMHH18].

## 2.6 Rendering Fundamentals Summary

Below is a short summary of concepts I introduced in this Chapter, along with references to their corresponding Sections, Figures, or Equations.

- I started by defining *rendering* (2.1) and outlining its key components, such as *object representation* (2.2), camera projection, lighting, and post-processing. I explained different methods of representing 3D objects in computer memory, focusing on *polygonal meshes* (2.2.1) for standard geometry and *volumetric representations* (2.2.2) for complex structures like clouds or fog.

- I introduced the main *vector spaces* (2.3) used in rendering, including the *object space* (2.3.1), *world space* (2.3.2), *camera space* (2.3.3), and *screen space* (2.3.4). The role of both *direct and indirect lighting* (2.4), together referred to as *global illumination* (2.4.2), was highlighted, emphasizing its importance for depth perception and realism of the final image.

- Furthermore, I related global illumination (GI) to *Kajiya's rendering equation* (2.4.4), which uses a mathematical framework called *Bidirectional Reflectance Distribution Function (BRDF)* (2.4.3) to represent light's interaction with real-world materials. This included an explanation of *diffuse* (2.10), *specular* (2.11), and *glossy reflections* (2.4.3), supported by their typical use cases and equations.

- Moreover, I described the *Monte Carlo integration* (2.4.5), which is used by many methods to approximate the rendering equation numerically and will be explored in greater detail in the next Chapter.

- Finally, the most important *light types* (2.5) were introduced, such as *directional lights* (2.5.2), *point lights* (2.5.3), *spotlights* (2.5.4), and *area lights* (2.5.5). This was complemented by highlighting the difference between *hard and soft shadows* (2.5.1).

While these concepts should be enough to understand Lumen's features and limitations, I recommend reading the book *Real Time Rendering* by Akenine-Möller et al. [AMHH18], as it contains many relevant details that are simply beyond the scope of this thesis.

# Chapter 3

# Illumination Computation Techniques

Various methods are used to calculate lighting in computer graphics. Some of them build upon the Monte Carlo integration approximation of the rendering equation and offer high-fidelity visuals at the cost of performance. This makes them useful for offline rendering, where quality is more important than speed. Other techniques can produce images in real time at the cost of omitting indirect lighting bounces or other crucial aspects of realistic lighting, such as plausible mirror reflections. There are methods which excel at both, but have their own drawbacks. Some of them, for example, lack dynamic adjustment to changes in the scene [Fel23, AMHH18, RDGK12].

In this Chapter, I will introduce the fundamental concepts of some of the most commonly used lighting techniques. I will mainly focus on *ray tracing* and *path tracing*, as they are the most widely used and influential. Next, I will talk about some optimization tricks used to accelerate these methods, as most of them are not only interesting on their own, but also highly relevant for Lumen. Finally, I will mention *radiosity* and *photon tracing*, since they also share some similarities with Lumen, which will be hinted at in Chapter 4.

If you are interested in a broader overview of more methods for calculating GI in real time along with their mutual comparisons, refer to Ritschel et al. [RDGK12] or Akenine-Möller et al. [AMHH18], where you can find details that are beyond the scope of this thesis.

## 3.1 Light Path Expressions

One important characteristic to study when discussing any lighting method is its ability to model various light paths. Heckbert's [Hec90] precisely defined regular *light path expressions (LPE)* are typically used to concisely describe which surfaces (i.e. diffuse or specular) the light interacts with on its way from a light source $L$ to the camera (eye) $E$ (or the other way around). These interactions may include reflection, transmission, or volume scatter.

The rendering equation (described in Section 2.4.4) can be expressed as $L(D|S|G)^*E$, which means that light bounces between any number of diffuse, specular, or glossy surfaces before reaching the camera. Modeling this expression is the goal we want to achieve for full global illumination.

For more information on this notation, please refer to Heckbert [Hec90] or Akenine-Möller et al. [AMHH18], where more examples are shown.

## 3.2 Rasterization

Instead of relying on the physically-based rendering process described in Section 2.4.2 where we shoot rays from the camera, most computer games use a different technique to produce 2D images from a 3D scene [AMHH18, FvBS05].

The core idea of this approach involves transforming all objects, represented as triangle data along with their modeling matrices, into screen space via a sequence of matrix multiplications, where each matrix represents a transformation from one vector space to another. Once the triangles are transformed, they are converted to discrete fragments in a process called *rasterization*. Afterwards, the vertex values used for shading and texturing are interpolated across the rasterized primitive. Direct lighting, expressed as $L(D|S)E$, is then calculated for each pixel.

In this Section, I will outline the individual components of this graphics rendering pipeline along with its advantages, disadvantages and possible extensions. For a more comprehensive and detailed description, please refer to Felkel et al. [FvBS05] or Akenine-Möller et al. [AMHH18].

### 3.2.1 The Rasterization Rendering Pipeline

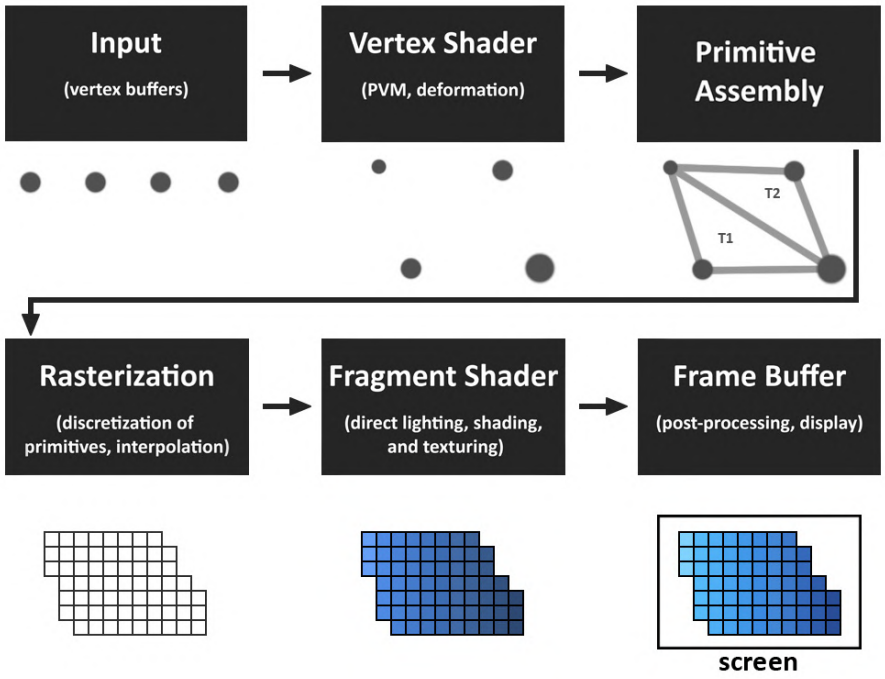



**Figure 3.1:** A simplified chart of the rendering pipeline (inspired by: [Cin23]).

The six main parts of the rendering pipeline can be seen in Figure 3.1 and are described below. Optional steps, such as the geometry shader, are omitted [AMHH18, Bra23].

- *Input* - The pipeline begins by sending vertex data stored in buffers along with the associated vertex and fragment shaders to the GPU.

- *Vertex Shader* - In this phase, each vertex is transformed from object space to world space through multiplication with a model matrix, then to camera space through multiplication with a view matrix, and finally projected to the normalized clip coordinates through multiplication with a projection matrix. These linear transformations are typically represented as a single all-encompassing matrix called *PVM matrix* (where P stands for projection, V for view, and M for model). All vertices outside the frustum specified by the projection matrix' clip range are discarded. Afterwards, a perspective division is applied. Note that this entire process occurs in a four-dimensional homogeneous coordinate system, as translation is not a linear transformation in 3D and therefore cannot be described as matrix multiplication.

- *Primitive Assembly* - During this stage, the vertices outputted by the vertex shader are grouped together to form primitives. While I previously mentioned triangles as the most commonly used type of graphic primitive, points and lines are also viable options. Additional buffers can be used to specify which vertices are connected together. This can be used to save memory, since vertices that are used to construct more primitives simultaneously need to be stored in memory only once.

- *Rasterization* - Rasterization converts the previously formed groups of primitives into fragments (one for each pixel). The normal, color, and texture values are interpolated across all pixels that belong to the surface of the previously formed primitive.

- *Fragment Shader* - Per-pixel lighting is calculated using a selected reflection model, such as Lambertian or Blinn-Phong, utilizing the interpolated values. Blending the resulting color with previous values in the framebuffer is also necessary to correctly display transparent and semi-transparent objects.

- *Framebuffer* - After all draw calls are completed, the resulting colors of each pixel are saved in the framebuffer. The depth of each pixel is usually stored separately in the Z-Buffer and can be used for depth sorting. Before transferring the buffer data to the display device and rendering the image, post-process effects such as bloom, color grading, or fast approximate anti-aliasing (FXAA) can be applied.

### 3.2.2 Advantages and Disadvantages

The main advantage of rasterization is its speed, which can be further enhanced by various optimization techniques invented over the last few decades. This includes, but is not limited to: *levels of detail*, *view frustum culling*, *occlusion culling* and *deferred shading*. All of these tricks are outside the scope of this thesis, but can be further studied in Felkel et al. [FvBS05] or Akenine-Möller et al. [AMHH18].

On the other hand, shading calculations are typically performed in parallel for each fragment in the screen raster, where information about nearby objects is not available. This implies that some desired physically accurate effects, such as shadows and additional lighting bounces, which require the full representation of the 3D scene, are not inherently part of this pipeline and must be integrated using additional techniques. For example, shadows can be calculated with *shadow maps*, *shadow volumes* or approximated by *screen space ambient occlusion (SSAO)*. Mirror-like reflections can be added by reusing the information stored in the framebuffer at the cost of being limited to screen information only. Similarly, global illumination for static parts of the scene can be precomputed offline, stored in lightmaps (described in Section 3.7) and added during the shading process.

Having all of these effects be solved with different, and in some cases limited techniques can make the whole process more conceptually complex than methods like *path tracing*, where accurate shadows and color bleeding are inherently outputted by the algorithm and scale better as the amount of lights in the scene increases.

An illustration of the quality difference between the same 3D scene rendered using Blender's rasterization algorithm, *Eevee*, and its path tracing engine, *Cycles*, can be seen in Figure 3.2. Note that shadows (not shading) are purposefully disabled in the Eevee version of the scene to further emphasize how visually lacking the output of a bare-bones implementation of the rasterization pipeline can be.

I believe that for many video-game players, having access to global illumination may not ultimately be worthwhile if the decrease in performance is too significant when compared to a method they have been accustomed to for decades. For this reason, I compare Lumen's performance with rasterization when testing Fortnite in Section 5.4. Similarly, in the following Sections of this Chapter, most of the described methods are compared to rasterization as well.

However, note that many of the methods mentioned further or their screen-space variants, including Lumen, can make use of some parts of the rasterization pipeline, such as the G-Buffer.

## 3.3 Ray Tracing

In this Section, I will discuss some variations of the well-known *ray tracing* algorithm (sometimes called a backward ray tracing, since the process is the

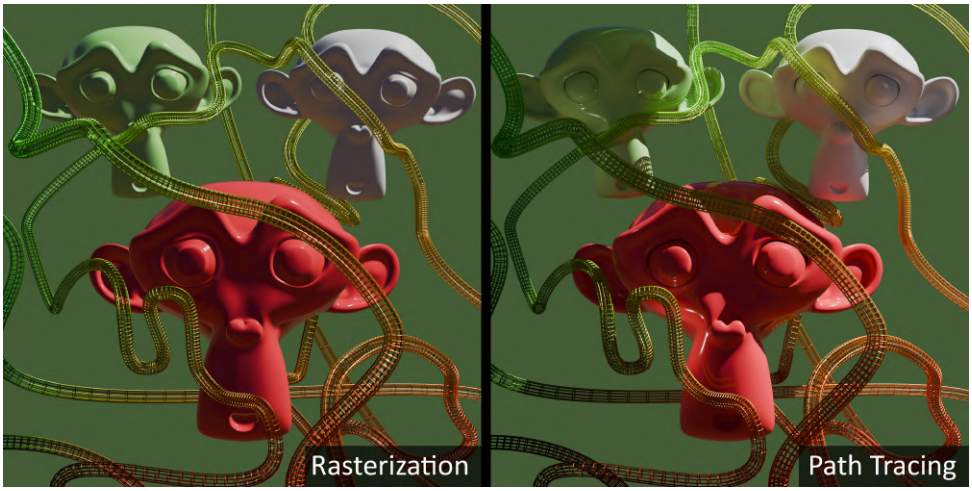



**Figure 3.2:** A comparison of a scene rendered using shadowless rasterization and path tracing.

inverse of how real-world light reaches our eyes) along with a few optimization techniques used to enhance its performance. As this method plays a pivotal role in Lumen's own light computations, more attention will be devoted to it in comparison with the methods I will mention afterwards.

### 3.3.1 Whitted Ray Tracing

As indicated in Section 2.4.5, *Whitted* ray tracing approximates the rendering equation by sampling the most crucial directions of the incoming radiance for a given point $x$. The basic implementation of this idea (illustrated in Figure 3.3) is that through every pixel on the screen, a single primary ray is sent from the camera. Each ray has an attribute $d$ that signifies its current recursive depth. At the start, we set the value of $d$ to 0 and follow the steps outlined below [Fel23, AMHH18].

1. If no intersection with a surface is found, return the background color. Otherwise, continue with Step 2.

2. Cast shadow rays in the direction of each light source (emissive materials are typically not taken into account in this step). If these rays intersect an object before reaching the given light source, the original surface is occluded, and thus shadowed. Otherwise, calculate the diffuse reflection if the material is diffuse and continue with Step 4. If the object is fully reflective, proceed with Step 3 instead.

3. If the recursive limit is not yet reached, send a reflection ray in the direction of the ideal specular reflection. Set its value of $d'$ to $d + 1$ and repeat steps 1 to 3. Similarly, another ray can be sent through the surface if it has a transmissive material. The direction of this ray will be based on the *index of refraction (IoR)* of that material. If the recursive depth is reached or a diffuse surface is hit, continue with Step 4.

4. Sum the contributions of all shadow, reflection and refraction rays and return the result. *Lambertian*, *Blinn-Phong* or *Cook-Torrance* shading models can be used to calculate the illumination by each light source at the given surface point, similarly as with fragment shader used in rasterization.

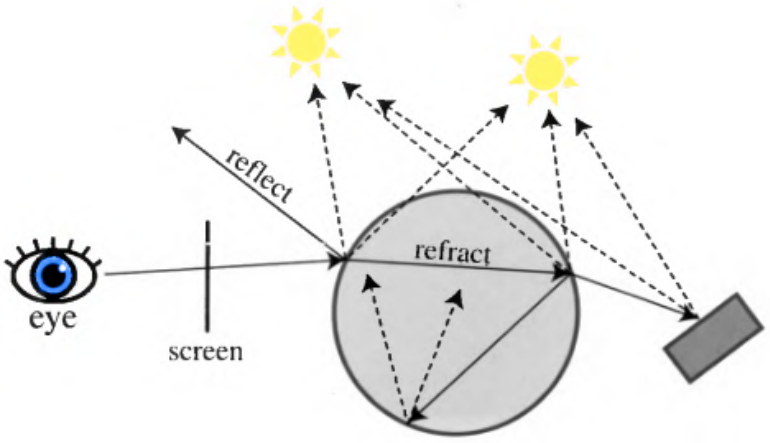



**Figure 3.3:** An illustration of the Whitted ray tracing algorithm (source: [AMHH18]).

Whitted ray tracing can be described using light path expressions as $LD?S^*E$. It produces sharp shadows and mirror reflections, which, while not perfectly realistic, serve as a solid foundation for its extended versions covered in the subsequent Sections. Furthermore, its physically based approach is more intuitive and less limited than techniques like shadow maps, which are used to approximate shadows in rasterized scenes.

Finally, since only the first surface each ray hits is considered, depth sorting is inherently solved by this algorithm, which is another advantage that ray tracing has over rasterization [AMHH18].

### 3.3.2 Distributed Ray Tracing

*Distributed* (also known as *distribution*) ray tracing is based on the Monte Carlo integration method described in Section 2.4.5. When a ray hits a surface, instead of reflecting it only towards the most important and specifically chosen directions, multiple secondary rays are sent in randomly selected directions (up to a certain recursion depth). Averaging their results allows for approximating diffuse inter-reflections, soft shadows, and even more advanced optical effects, such as depth of field or motion blur. The ideal GI light path expression described in Section 3.1 can be achieved using this technique [AMHH18, Jus20a].

Note that when using this technique, the number of rays increases exponentially with each secondary ray. Unfortunately, for a noise-free image, a large number of secondary rays may be required.

### 3.3.3 Path Tracing

Path tracing is similar to distributed ray tracing, with one major exception. Instead of shooting many secondary reflection rays in each recursive call of the algorithm, we cast only one secondary ray in a random direction around the hemisphere of the surface, unless the recursive depth has been reached. To compensate for this very naive approximation of the *illumination integral*, we send a significantly higher number of primary rays for each pixel and average their results. We refer to the number of sent primary rays as the number of samples. To achieve smooth, noise-free images, the number of required samples (with uniform distribution) is typically around a thousand, although this value heavily depends on the ray recursion depth, the selected sampling strategy, and the complexity of the scene.

Figure 3.4 shows the difference in image quality between two different scenes rendered with various sample counts. As expected, the bottom scene, with more complex lighting and materials, is noisier in all cases. The maximum number of secondary light bounces was set to 4 for both diffuse and specular reflections [Kim22b, Jus20a, Fel23, AMHH18].

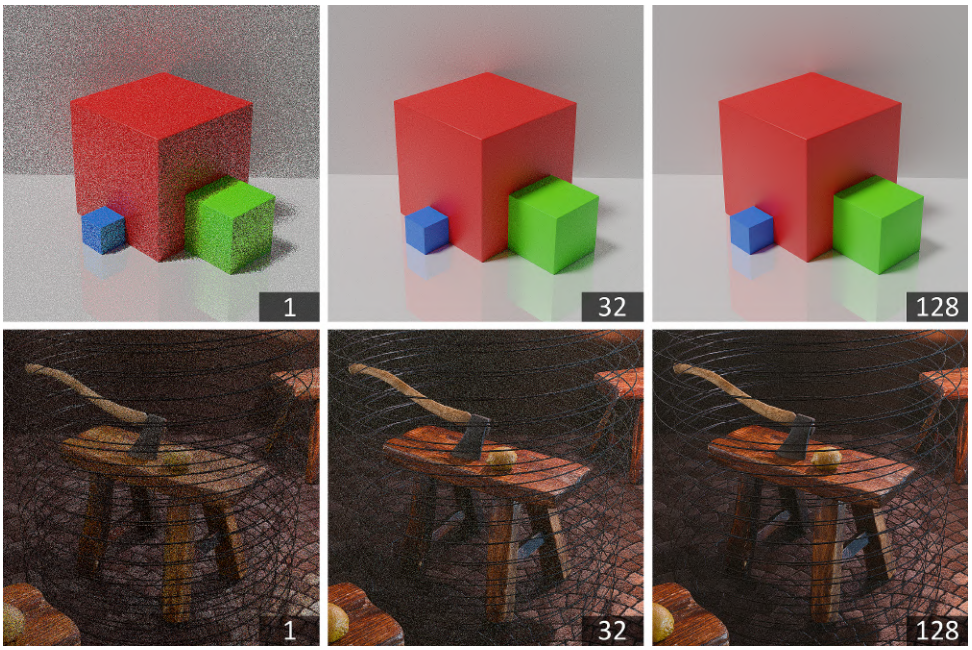



**Figure 3.4:** A comparison of two differently complex scenes rendered in Blender using the Cycles path tracer with 1, 32, and 128 samples.

## 3.4 Ray Tracing Optimizations

Without any acceleration structures, the time complexity of Whitted ray tracing can be roughly described as $O(n \cdot m)$ as it scales linearly with both the number of pixels on the screen $n$ and the number of triangles or other

primitives in the scene $m$. Stochastic ray tracing methods introduce an additional scaling factor, making these techniques even slower.

In this Section, I will introduce the most commonly used techniques which allow ray tracing algorithms to trace fewer rays and find their intersections with geometry faster. Most of the optimizations covered in this Section, albeit only on high level, will provide a foundation to later understand how Lumen achieves its presumably noise-free results in real-time [FvBS05, AMHH18].

### 3.4.1 Importance Sampling

In Section 2.4.5, the general Monte Carlo approximation (2.13) was introduced, which can estimate the value of a given definite integral by random sampling. In the specific case of estimating how the incoming radiance reflects of a surface point $x$ in direction $\omega_o$, we need to approximate the *Illumination Integral* 3.1, which was introduced in Section 2.4.4 as the most important part of the rendering equation [Kri, BMDS19].

$$\int_{\Omega} L_i(x, \omega_i) BRDF_r(x, \omega_o, \omega_i) \cos\theta_i d\omega_i \tag{3.1}$$

The formula for approximating this integral can be written as follows:

$$\lim_{N\to\infty} \frac{1}{N} \sum_{i=1}^{N} \frac{L_i(x, \omega_i) BRDF_r(x, \omega_o, \omega_i) \cos\theta_i}{PDF(\omega_i)} \tag{3.2}$$

The value $N$ represents the amount of samples. The computed result for each sample is divided by its probability, allowing us to compensate for the sample's bias. This is typically represented by *PDF*, which stands for *probability density function* [Wri21].

In the example from Section 2.4.5, the PDF was omitted since we used a uniform distribution, where each sample had the same probability of being chosen. However, a uniform distribution is not always optimal, as samples with lower values of the integrated function contribute less to the final averaged result, leading to a higher variance. This, in turn, results in increased noise in the specific case of algorithms such as path tracing or distributed ray tracing.

This strategic biasing towards more important samples is called *importance sampling*, and is commonly used not only in path tracing and similar techniques, but also in Lumen. When implemented correctly, it allows us to trace fewer rays per pixel, thereby increasing the rendering speed. To achieve completely noise-free images without further denoising, the PDF must be perfectly proportional to the integrand [BMDS19].

In ray tracing, the most commonly used biases for importance sampling are:

- biasing towards directions with higher incoming radiance
- biasing towards directions with larger value of BRDF
- biasing towards directions to brighter and closer light sources

- biasing towards directions closer to the surface's normal, as their cosine value will be higher
- biasing towards directions which have previously proven to have higher values of the sampled function

A more sophisticated approach is to use *multiple importance sampling*, which samples the integral using multiple distributions and cleverly weights their results (for example, by using a specific heuristic).

Examining importance sampling in greater detail is beyond the scope of this thesis, but the way Lumen implements it is described in Section 4.8.1, where its effect on noise reduction is shown in Figure 4.25.

For further reading on this topic, please refer to Bako et al. [BMDS19] or other sources [Mut22, rav20, WNK].

### 3.4.2 Bounding Volume Hierarchy

In the most basic implementation of ray tracing, finding ray intersections with the scene's geometry introduces the aforementioned linear scaling factor. That is because we need to check every triangle one by one until we find the first which the ray intersects.

This can be significantly accelerated by enclosing connected triangles and subsequently whole objects and groups of objects in conservative bounding volumes. This allows us to first check for an intersection with each such volume. If the ray misses a given volume, there is no need to check the subsequent volumes, objects, or geometry contained within it, as we know that the ray cannot intersect them [Jus20b, FvBS05, NVId].

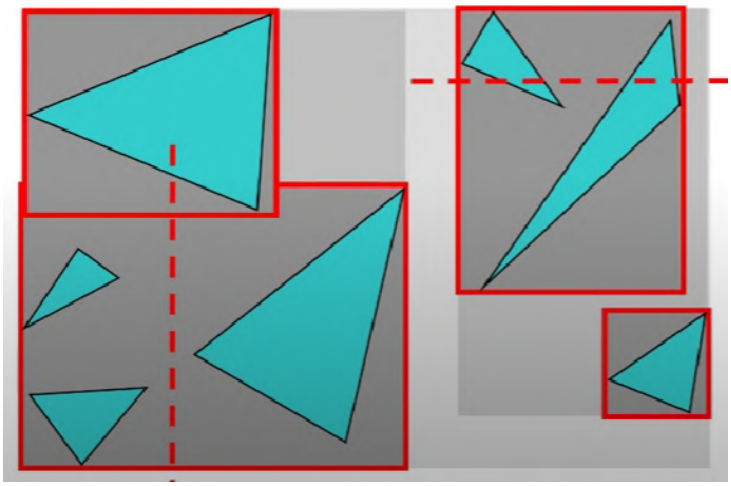


**Figure 3.5:** An illustration of a bounding volume hierarchy (source: [Jus20b]).

The time and memory complexity of constructing and using this so-called *bounding volume hierarchy (BVH)* depends on the chosen type. For example, axis-aligned bounding volumes are faster to build, but less precise to trace against. Furthermore, if the scene is not static, the BVH structure may

need to be rebuilt dynamically as the objects move around, which can be computationally expensive.

#### Advantages and Disadvantages of BVH

While BVHs are relatively easy to understand, construct, and traverse, they are not without flaws. The bounding volumes at the same tree level can overlap, which means that finding a ray intersection with one of them does not inherently guarantee that others do not need to be traversed as well.

This is true because triangles contained within the other bounding volumes can be closer to the camera than triangles contained within the nearest bounding volume. This problem is illustrated (in a simplified 2D form) in Figure 3.6 and indicates that tracing rays against scenes with many overlapping objects can be very expensive even when using BVH. Lumen's hardware ray tracing (described in Section 4.6) suffers from this exact issue.

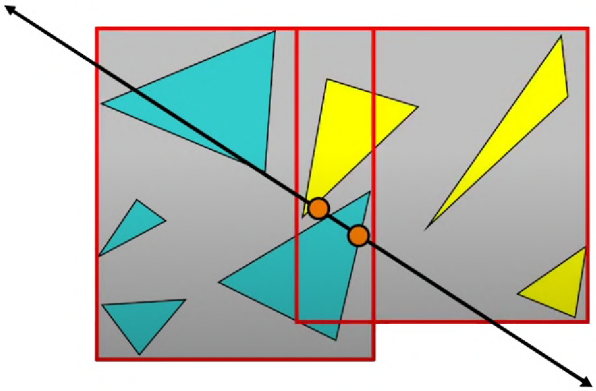


**Figure 3.6:** An illustration of a problem with BVH (source: [Jus20b]).

One possible solution is to use a different acceleration structure in place of BVH, such as *KD trees*, *Octrees* or *BSP trees*. However, those come with their own sets of drawbacks, namely a lack of support for dynamic objects or slower rebuilding, and as such, BVHs are still the most commonly used acceleration structure for ray tracing. For more information on acceleration structures, please refer to Solomon [Jus20b] or Felkel et al. [FvBS05].

### 3.4.3 Hardware Acceleration

Graphics processing units are well-suited for accelerating the calculations used in real-time rendering, as they allow for very fast parallel processing. Ray tracing is no exception, since the resulting color of each pixel is independent on the results obtained by rays sent through other pixels, indicating that all primary rays can be computed in parallel.

However, the GPU support for tracing rays does not stop there, as many of the new generations offer additional specialized components to make this process even faster. In this Section, I will briefly mention some of the

features NVIDIA's Turing GPU architecture used in their RTX cards to enable noise-free ray tracing in real time. Note that AMD's improvements in their RX series are conceptually similar and will therefore not be covered here [NVI20, NVId].

One pivotal improvement introduced in Turing architecture is the inclusion of new *RT cores*, which contain two specialized units. The first performs bounding box tests, which significantly accelerates the BVH (Section 3.5) traversal. The second unit performs ray intersection testing with geometry primitives. This frees up thousands of instructions slots the streaming multiprocessor can spend on calculating other important parts of the pipeline [NVI20, NVId].

The other Turing feature which enables ray tracing with higher frame rates, although not directly related to it, is the introduction of faster *tensor cores*. These GPU cores are specialized in large-scale matrix multiplications typically used in machine learning. In the context of rendering, they are suitable for AI-driven upscaling of the image. One notable example of AI upscaling that relies on tensor cores is NVIDIA's DLSS, which stands for *Deep Learning Super Sampling*. This is a temporal anti-aliasing upsampling algorithm which allows us to natively render at lower resolution, then upscale the frame to the desired resolution. This gives us the option to cast significantly fewer rays than if we would render the scene at full resolution. For more information on this topic, please refer to NVIDIA's website [NVIa, NVId, NVIb].

Finally, tensor cores can be used for AI-accelerated denoisers, which is further explained in the following Section.

Note that the most commonly used interface that allows for efficient use of modern GPUs specifically for ray tracing is Microsoft's *DirectX RayTracing (DXR)*. Unreal Engine 5 uses DXR for its path tracing and also for Lumen's hardware ray tracing [WNK]. Its absence from previous generations of GPUs compelled Epic Games to develop their own software ray tracing pipeline. This is further described in Sections 4.1 and 4.5.

### 3.4.4 Screen Space Denoising

As mentioned in previous Sections, the quality of an image rendered using Monte Carlo ray tracing methods heavily depends on the number of samples. However, if we want to use these techniques for real-time rendering, shooting tens, hundreds, or even thousands of rays per pixel is unattainable. Tracing one path per pixel is typically the upper limit [Kim22a, NVI17, WN23]. This results in very noisy images, which must be further processed to achieve plausible results. There are three main categories of screen space denoisers, but modern approaches often incorporate elements from more than one category:

- **Spatial filtering** - This approach utilizes data from the current framebuffer, typically averaging, interpolating, or extrapolating each color value to neighboring pixels using convolution matrices. While this may

function correctly in some cases, it often introduces undesirable blurring or loss of detail, and as such, additional context in the form of depth and normals may be required. Some examples of spatial filters are Gaussian filters, low-pass filters, bilateral filters, and wavelet filters [FvBS05, Kim22a].

- **Temporal reuse** — These techniques utilize data from previously rendered frames, often averaging their pixels with those from the current framebuffer. Note that without careful reprojection using motion vectors or other heuristics, this approach can cause ghosting artifacts, where data from previous frames influences the current samples at incorrect locations. Another issue potentially caused by relying on temporal accumulation is the slow propagation of light changes. Unfortunately, as shown in Section 5.2.3, Lumen also suffers from this problem. Examples of denoisers that use temporal reuse include SVGF ([NVI17]) and TAA [Dig24, WN23, Wri21, Kim22a].
- **AI-driven denoising** — This approach, implemented in denoisers such as NVIDIA's Optix ([NVIc]) relies on machine learning and may use both spatial and temporal filtering.

Lumen also uses denoising, which is described in Section 4.8.

### 3.4.5 Real-time Ray Tracing

Unfortunately, even with implementing the optimization techniques covered in this Section, Monte Carlo ray tracing methods are usually still too slow to be used in real time applications, especially if very high frame rates and screen resolutions are targeted. This is not only because the number of cast rays increases exponentially and to achieve noise-free images, a huge number of them is needed, but also due to the fact that finding ray intersections with the scene's geometry is very time consuming.

The Whitted ray tracing is at the present the preferred solution for high-quality real-time local illumination, mirror reflections, and refractions in video games, especially when combined with neural GPU-accelerated image upscaling such as the aforementioned *NVIDIA DLSS*. The popularity of these methods may increase as newly released GPUs continue to get faster and more accessible to a wider audience [Jus20b, KG09].

## 3.5 Radiosity

Radiosity was the first rendering technique used to simulate inter-reflections between fully opaque diffuse surfaces. To calculate the outgoing radiance $L_o$ at each surface point, we first compute the *radiosity* (which represents the total amount of light energy leaving a surface per unit area, regardless of direction) for the whole surface (represented as a rectangular *patch* of arbitrary size). This is illustrated in the following Equation [AMHH18, GTGB84]:

$$B_i = E_i + \rho_i \cdot E \tag{3.3}$$

Here, $B_i$ represents the radiosity of a surface patch $A_i$ and is equal to the emitted energy $E_i$ plus the reflected irradiance $E$. The proportion of this reflected irradiance is determined by the reflectance $\rho_i$ of the surface $A_i$, which is a real value between 0 and 1. Irradiance can be expressed as the sum of the radiosity reaching the surface patch $A_i$ from all the surface patches in the scene. This relation can be seen in the Equation below.

$$B_i = E_i + \rho_i \sum_{j=1}^{n} F_{ij} B_j \tag{3.4}$$

Here, $n$ stands for the number of patches in the scene and the *form factor* $F_{ji}$ indicates what portion of the radiosity $B_j$ from surface $A_j$ reaches the surface $A_i$. This value depends on the area of these surfaces, their relative distance, and their orientation. It can be calculated by evaluating the following integral, which is illustrated in Figure 3.7:

$$F_{ji} = \frac{1}{A_j} \int_{A_j} \int_{A_i} \frac{cos\phi_j \cdot cos\phi_i}{\pi r^2} h_{ji} dA_j dA_i \tag{3.5}$$

Here, $r$ represents the distance between patches $A_i$ and $A_j$ and $\phi_i$ and $\phi_j$ are the angles between their normal vectors and the distance vector. The visibility factor $h_{ji}$, which ranges from 0 to 1, is also essential to model occlusion. If there is another surface $A_h$ between patches $A_i$ and $A_j$, then the form factor is reduced by the portion of $B_J$ that is blocked by $A_h$.

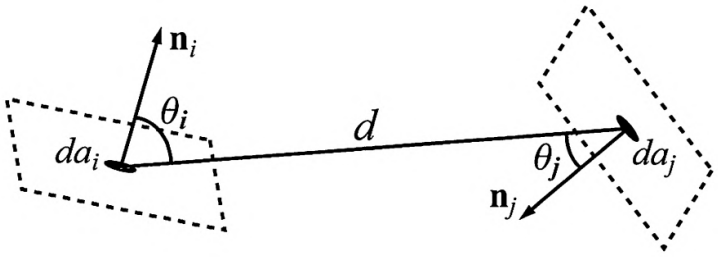



**Figure 3.7:** A visualization of attributes used to calculate a form factor $F_{ji}$ (source: [AMHH18]).

Once we have all the form factors, we can think of calculating the radiosity of each surface patch as solving a linear system of equations. This can be represented by the following matrices:

$$\begin{pmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{pmatrix} = \begin{pmatrix} 1-\rho_1 F_{11} & -\rho_1 F_{12} & \cdots & -\rho_1 F_{1n} \\ -\rho_2 F_{21} & 1-\rho_2 F_{22} & \cdots & -\rho_2 F_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ -\rho_n F_{n1} & -\rho_n F_{n2} & \cdots & 1-\rho_n F_{nn} \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{pmatrix} \tag{3.6}$$

After solving this linear system of equations, we will have a value for the radiosity $B_i$ for each surface patch $A_i$. However, calculating the solution is

computationally expensive, so an iterative process is typically used instead. In this approach, we first set the emittance values for all emissive and directly illuminated patches, then calculate $B_i$ $n$ times for every pixel using the Equation 3.4, where $n$ is the number of diffuse light bounces we are interested in. Typically, the radiosities converge after only a single-digit number of iterations, which makes this technique usable for real-time applications. Iterative radiosity models each light path $LD\{n\}E$.

Once we have a sufficiently accurate approximation of radiosity $B_i$ we can calculate the radiance $L_o$ simply by dividing $B_i$ by $\pi$. To obtain the radiance at each pixel, we can linearly interpolate the values between patches. This is illustrated in Figure 3.8. For more details on the math behind these calculations, please refer to Goral et al. [GTGB84] or Burenius [Bur09], where they are explained in greater detail.

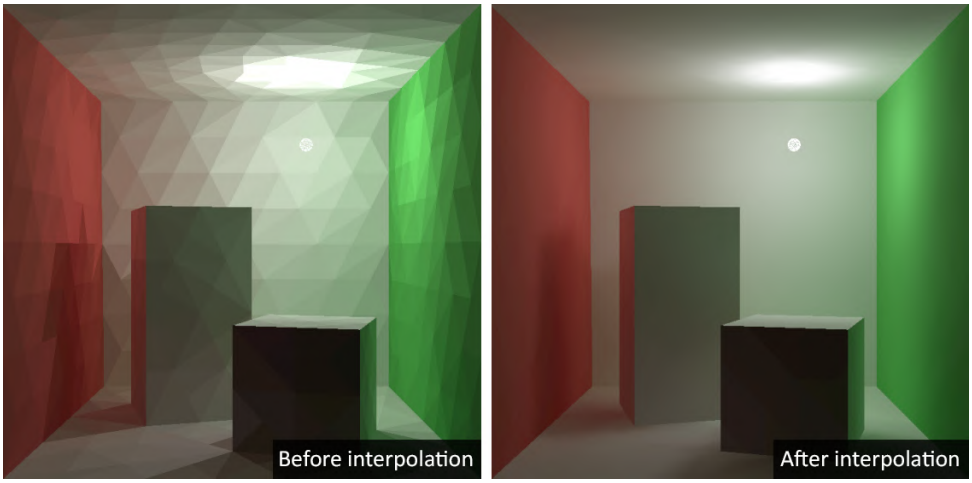



**Figure 3.8:** A scene rendered using radiosity, before and after interpolation (source: [Bur09]).

Radiosity is typically used for offline rendering, primarily for smaller architectural visualizations, as it has a time complexity of $O(n^2)$, where $n$ is the number of patches. This makes it less suitable for larger scenes with many patches. Several real-time variations of the technique exist, such as the one described by Burenius [Bur09].

However, Radiosity's inability to produce specular reflections severely limits its use cases in video games or projects where a complete global illumination is desired. Additionally, form factors must be recalculated as the objects in the scene move, which makes this technique limited to static scenes only.

These limitations show that speed and simplicity are not the only qualities we desire from a real-time GI solution.

## 3.6 Photon Mapping

Another method which is interesting to compare with Lumen is called photon mapping. Developed by Henrik Wann Jensen [WJ05], this technique separates lighting calculation into two passes, similarly to Radiosity.

First, a large number of rays (representing photons) are shot from the light sources into the scene, where they are reflected, transmitted, or absorbed during intersection with surfaces based on their material properties. At each such intersection point, a radiance data is stored in a so-called photon map.

Once the photon map is constructed and efficiently stored, a second rendering pass is performed, called *the final gather* or the *the final gathering*. Here, we typically use ray tracing to compute direct lighting for each pixel and then utilize the data stored in nearby positions in the photon map to estimate the indirect lighting, instead of relying on secondary bounces as Monte Carlo methods do [WJ05, AMHH18].

Similarly to radiosity, photon mapping is less suitable for real-time dynamic global illumination, though it works with more general light paths, such as $L(D|S) + E$. Specifically, photon mapping is widely used for offline rendering of caustics, as Monte Carlo methods generally require a very large number of samples to render them effectively. Unfortunately, Lumen, while similar to photon mapping in being separated into multiple passes, does not support caustics for many reasons, such as its low ray budget per pixel and reliance on spatial and temporal filtering [WJ05, Cor22, Gam23].

## 3.7 Lightmap Baking

Lightmap baking is a frequently used method for both real-time global and local illumination, which accelerates the entire process by dividing the scene into two parts: static and dynamic.

All lighting for the static part of the scene is precalculated and baked into light-map textures (typically with path tracing, though any of the aforementioned GI methods can be used), and only the lighting for the dynamic part of the scene is calculated in real time as the camera, objects, or light sources move.

The disadvantages of this method are discussed in Section 4.1 where they are identified as one of the main reasons Epic Games decided to create Lumen in the first place. For further reading on lightmap baking, please refer to Akenine-Möller et al. [AMHH18] or the Epic Games' official documentation [Gam24a].

## 3.8 Illumination Techniques Summary

In this Chapter, I introduced some of the most relevant techniques used for computing illumination in computer graphics. Below is a short summary of

these methods, along with references to their corresponding Sections, Figures, or Equations.

- First, I outlined the basic *rendering pipeline* (3.1), which uses a nonphysically-based approach called rasterization to convert objects within a 3D scene into pixels on our screen. I discussed its advantages and disadvantages (3.2.2) for real-time rendering and highlighted its speed and efficiency, while also pointing out its general inability to produce full global illumination without relying on precomputed *lightmaps* (3.7).
- Next, I discussed various types of ray tracing (3.3) approaches, such as *Whitted ray tracing* (3.3.1) or *path tracing* (3.3.3), along with many optimization techniques used to increase their rendering speed and efficiency, focusing on those relevant to Lumen. Namely, I covered *bounding volume hierarchies* (3.5), *denoising* (3.4.4), *importance sampling* (3.4.1) and recent innovations in *hardware acceleration* (3.4.3).
- Finally, I mentioned other well-known techniques that share certain similarities with Lumen, specifically *Radiosity* (3.5) and *photon mapping* (3.6). Furthermore, I specified their typical use cases, limitations, and advantages they have over Lumen, with simplicity being the most prominent one.
- Throughout this Chapter, I used *Heckbert's light path expressions* (3.1) to compare the ability of all these methods to simulate various light paths.

Understanding these techniques helped me tremendously to analyze, appreciate, and contextualize Lumen's technical details, which I describe in the following Chapter.

Be warned that Lumen has many interconnected and heavily configurable layers with multiple purposes, making it significantly harder to fully grasp at first compared to methods mentioned in this chapter.

# Chapter 4

# Lumen Analysis

In this Chapter, I will present Epic Games' motivation behind creating Lumen, along with its current capabilities.

Afterwards, I will describe the main ideas that brought this technology to life, followed by a detailed analysis of Lumen's individual components. This includes the types of ray tracing it supports and various optimization techniques used in different stages of its pipeline.

Lastly, I will briefly mention Lumen's settings and configurations from the user's perspective, as understanding them will be important during testing in the following Chapter (5).

## 4.1 Motivation

According to Daniel Wright, a graphics engineer at Epic Games, they set out to create Lumen as a solution to three important problems they identified with some methods commonly used to implement global illumination in real-time applications: lightmap baking (described in Section 3.7) and irradiance fields, which interpolate precomputed irradiance from probes placed within the world to nearby pixels [Wri21].

First, the idea of true dynamic real-time global illumination could create opportunities for game designers to explore new gameplay ideas, such as environmental destruction. This is challenging to achieve when relying on precalculated lighting, as it often causes a visual mismatch between the static and dynamic part of the scene. Irradiance fields typically propagate disocclusion changes, but not instantly, which can lead to visible artifacts.

Second, lightmap baking can be a very tedious process for the artists involved, as it can take up to many hours to bake the GI to an acceptable level of quality. Irradiance fields have a similar problem - to avoid artifacts such as light leaking, manual placement of probes is often necessary [WN23].

Finally, relying on lightmaps has an additional memory overhead during rendering, because the baked lighting must be stored in textures. This can be a significant problem in large open-world scenes.

The solution Epic Games came up with relies on Monte Carlo ray tracing (introduced in Section 2.4.5) enriched by many different optimization tricks, some of which build upon those introduced in the previous Chapter. The

basic overview of these concepts is provided in Section 4.3 and I further explain the individual parts of Lumen in their corresponding Sections.

### 4.1.1 The Hybrid Pipeline

When developing Lumen, the use of ray tracing seemed to contrast with Epic Games' other requirement: the global illumination had to run not only on the next generation of consoles and powerful computers with modern GPUs, but also on older hardware which does not support DXR. Furthermore, to satisfy demands for different types of projects, Lumen needed to support both large open worlds and smaller indoor scenes, targeting a minimum of 60 frames per second (FPS) when using the full HD resolution [Gamb, Gamc].

To fulfill both of these needs simultaneously, Lumen supports two types of ray tracing: software and hardware. *Software ray tracing* (described in Section 4.5) is the default option which supports a much wider variety of devices at the cost of imperfect visual quality. In contrast, *hardware ray tracing* (described in Section 4.6) can run on newer GPUs only, but allows for a more precise global illumination, including multi-bounce mirror reflections [Gamb, Gamc].

## 4.2 Capabilities and Limitations

In Unreal Engine 5.5, Lumen offers real-time global illumination and reflections. Both of those systems can be toggled on and off independently. Below is a more specific list of features that Lumen's GI currently supports: ([Gam23, Gama]):

- diffuse indirect lighting with infinite bounces for static meshes,
- sky lighting and shadowing,
- lower-quality GI for volumetric effects,
- light propagation for two sided foliage,
- all light types, including emissive materials,
- material ambient occlusion,
- soft shadows.

Based on which ray tracing pipeline the user decides to use, Lumen may have limitations which should be carefully considered. Some of them are listed below, with full official description available as part of the Engine's documentation [Gamc].

- Lumen's global illumination is not compatible with forward shading and precalculated static lighting stored in lightmaps.

- Software ray tracing does not support dynamic meshes (i.e. those with dynamic geometry), world position offset in materials, and walls thinner than 10 cm. More disadvantages of software ray tracing can be found in Section 4.5.8.
- The performance of hardware ray tracing is severely hindered by object overlaps.

## 4.3 High-Level Overview

My understanding of Lumen's complex hybrid ray tracing pipeline [Gamc, WNK, Gamb] is schematically shown in Figure 4.1 and conceptually described below.

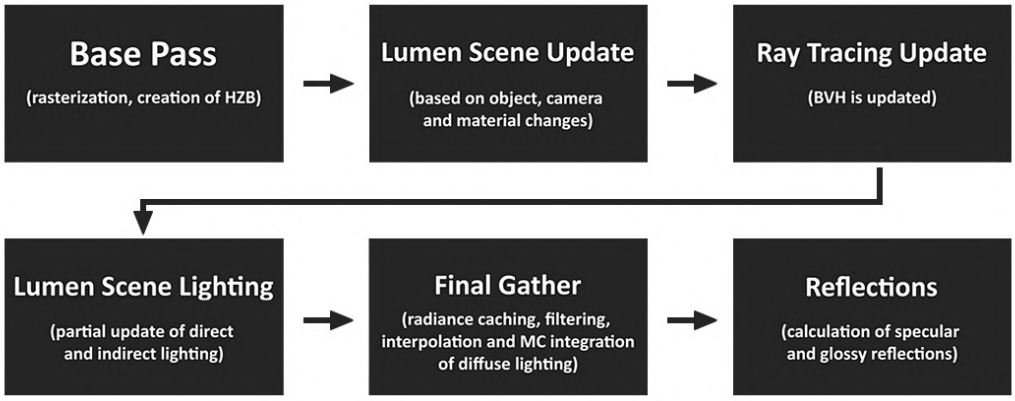



**Figure 4.1:** A high level overview of Lumen's rendering pipeline.

Note that all of its unique parts (such as the surface cache or the final gather) are analyzed in greater detail in their corresponding Sections. In addition to the resources listed below, I also used the Unreal Engine 5.3's source code (which is publicly available on GitHub) and the *Unreal Insights* profiler to better understand the pipeline.

Despite my best efforts, some details may have eluded me due to the sheer amount of used techniques, tricks, and their interconnectedness. Furthermore, I was unable to find default values for some of the attributes which were never explicitly defined in the code, nor mentioned in the external descriptions.

For official information about the concepts behind Lumen, please refer to the following sources: [Wri21, Unr21, WNK, Gamc, Gamb].

1. **The (Nanite) Base Pass** — The scene is rasterized. Depth, albedo (the diffuse color), normal, and material values are stored in buffers. Afterwards, a *hierarchical z-buffer* (HZB) is prepared for later use.

2. **Lumen Scene Update** — The surface cache (a texture atlas containing a parametrization of objects near camera) is updated based on spatial,

occlusion and material changes. Similarly, the mesh distance field (MDF) representation of the scene is updated if software ray tracing is used.

3. **Ray Tracing Update** — The ray tracing acceleration structure (two-level BVH) is updated if hardware ray tracing is used.

4. **Lumen Scene Lighting** — Direct lighting is partially updated for the surface cache using a fixed texel budget. For indirect lighting, the surface cache reads from itself and updates the values of the most important and outdated texels. These feedback-based changes are accumulated over multiple frames and converge to the approximation of the entire diffuse lighting path *final gather*.

5. **Final Gather** — The lighting stored in the surface cache is propagated to the pixels on the screen using ray tracing. Various optimization tricks, such as heavily downsampled screen and world space radiance caching, spatial and temporal reuse, and product importance sampling, are used to obtain noise-free results on a very small ray budget. The final gather for opaque surfaces is described in Section 4.8. Similar approach is used for getting the indirect lighting on volumetric objects [WNK].

6. **Lumen Reflections** — Reflections are calculated using a dedicated pipeline described in Section 4.9, which handles materials with different roughness values separately.

This multilayered approach has three following goals, all of which help to lower Lumen's rendering time without substantially compromising the visual quality.

- *Send the lowest possible number of rays while still maintaining a temporarily stable and noise-free image.* This is solved by the final gather's combination of screen space and world space radiance caching, along with lots of spatial and temporal filtering.

- *Find the intersections with objects quickly, without a substantial loss of precision and quality.* The software ray tracing variant uses mip-mapped distance fields to achieve this. Hardware ray tracing still traces against the original geometry, but uses a two-level BVH and a far field representation of the distant scene to accelerate the intersection search.

- *Reuse as much information as possible.* This is the primary function of the surface cache, but that is not the only example of information reuse. Lumen also incorporates spatial and temporal filters during many parts of its pipeline. Similarly, reflections reuse data stored in final gather's screen space and world space radiance cache.

Note that some of the used data structures, such as the *signed mesh distance field* representation for software ray tracing or the surface cache's cards are precomputed and individually stored during mesh import. However, this is an automatic process and requires no input from the artist except for some very specific edge cases covered in Sections 4.4, 4.5.2, and 4.7.2.

### 4.3.1 Ray Tracing in Lumen

As was already established, Lumen supports two ray tracing pipelines: software and hardware. The choice between these two changes the object representation the rays are traced against, impacting the performance and visual quality of the produced image. However, both pipelines have the same starting points (screen tracing) and end points (surface cache sampling), which is shown in Figure 4.2.

First, each ray is traced against the hierarchical z-buffer (HZB) (this data structure is described in Section 4.4). The screen space phase ends when each ray either travels a long enough distance, hits a pixel, exits the screen, or disappears behind a surface. Afterwards, the pipeline continues by tracing against either signed distance fields or the raw 3D geometry based on the selected ray tracing method. If a hit is registered, the corresponding value in the surface cache (used in both SW and HW ray tracing) is sampled. Otherwise, the ray returns the skylight color.

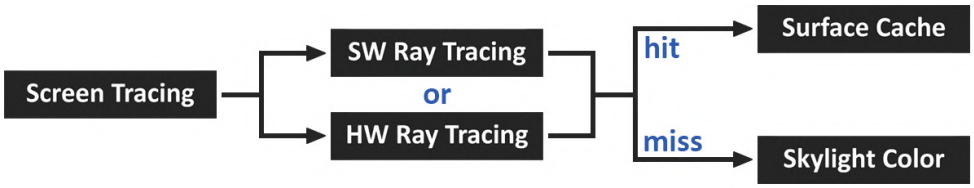



**Figure 4.2:** Lumen's simplified ray tracing scheme.

Note that both software and hardware ray tracing further divide the ray's lifetime into two stages, meaning that each ray is traced against three different scene representations in total. However, switching the representation on a per-ray basis would introduce a significant amount of overhead. Therefore, as already indicated, the process is globally divided into phases. During each phase, the ray continues from the position it ended at in the previous phase.

Furthermore, keep in mind that Lumen does not trace rays from the individual pixels on screen, but rather from probes placed either on the surfaces of objects or near them. Section 4.8 goes over this approach in greater detail.

## 4.4 Screen Tracing

Lumen relies on screen tracing (that is, tracing against the depth-buffer) to fix two main issues caused by the alternative scene representations used in the later stages of the pipeline.

First, software ray tracing uses a volumetric signed distance field representation of the objects for finding ray intersections. This can produce suboptimal results (such as light leaking or over occlusion) for very complex meshes that cannot be accurately described by voxel grids with limited resolution.

Hardware ray tracing faces similar limitations when using Nanite, as the dynamically culled and streamed meshes cannot be utilized by *DXR*, which is necessary for Lumen's BVH. This limitation is further described in Wright's SIGGRAPH 2022 presentation [WNK] and is currently addressed by relying on pre-generated *fallback meshes*.

A fallback mesh is a very aggressively decimated approximation of the original Nanite mesh. By default, fallback meshes contain only around 1% of the original vertices, although their complexity can be manually adjusted. While this allows Nanite meshes to be used for hardware ray tracing, visual artifacts may appear if the fallback is too sparse and lacks the individual refinement by an artist. An example of such mesh that could potentially cause problems for Lumen is shown in Figure 4.3.

Second, the distance-field representation does not support skinned and morph-able meshes, and neither does Nanite. Although this is expected to change in future releases (and Unreal Engine 5.5 already introduces Nanite support for skeletal meshes as an experimental feature), Lumen needed a solution for the time being. Otherwise, it would be barely usable for any project that contains dynamic meshes, such as animated characters [WNK, Gam24b, Gamc, Sch24, Sko23, Unr21]).


**Figure 4.3:** An illustration of a difference between a (on the left side) and its fallback variant (on the right side) used for hardware ray tracing, along with the corresponding triangle counts (source: [WNK]).

Lumen's screen tracing not only mitigates both of these issues, but is also faster and more memory-efficient for complex scenes, as it uses a stack-less walk of the rasterized depth map stored in the z-buffer [WNK]. This iterative process is further optimized by dynamically switching to coarser mip levels if a ray misses, thus reducing the average number of steps needed to find an intersection. The hierarchical z-buffer approach was inspired by Wolfgang Engel's method called *Hi-Z Screen-Space Cone-Traced Reflections* introduced in his book *GPU Pro 5*. Its more detailed and refined version can be found in the following article: [Lee21]. A visualization of tracing against HZB is

illustrated in Figure 4.4 [WNK, Sko23].

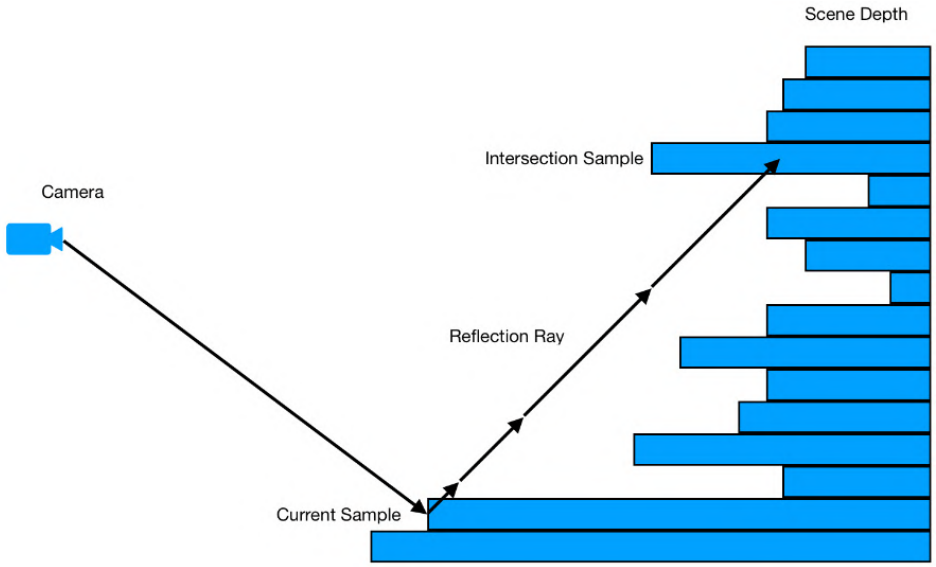



**Figure 4.4:** An illustration of HZB traversal. In this example, the ray misses during the first step and switches to larger steps by changing the Z-buffer mip level. This occurs again after the second miss, as indicated by the increasing length of the ray's segments (source: [Lee21]).

Lumen extends this technique by halving the base z-buffer resolution for diffuse rays and limiting the iteration count for rays at steeper angles, which are less likely to intersect nearby geometry. There is also an additional data compaction step after all rays are evaluated, which uses prefix sums to preserve ordering of nearby rays. Afterwards, the unsolved rays are passed to the next stage, which is either the software ray tracing (described in Section 4.5) or the hardware ray tracing (4.6) [WNK, Unr21].

One limitation of screen space rays is that they are not suitable for long-distance tracing, as they are very likely to go out of the screen or behind a surface. Nevertheless, they are a useful part of Lumen's hybrid ray tracing pipeline for the aforementioned reasons [WNK, Sko23].

## 4.5 Software Ray Tracing

Software ray tracing is one of Lumen's primary innovations. At a high level, it works by pre-generating a *signed distance field* representation of each mesh during import, which is then instanced and traced against using *sphere tracing*. If the ray does not intersect any mesh distance field within the first 2 meters, it is further traced against a global, less precise distance field instead. This structure (called *Global Distance Field* is dynamically updated as objects in the scene move [WNK, Sko23, Unr21].

The entire tracing pipeline when using software ray tracing is schematically shown in Figure 4.5.

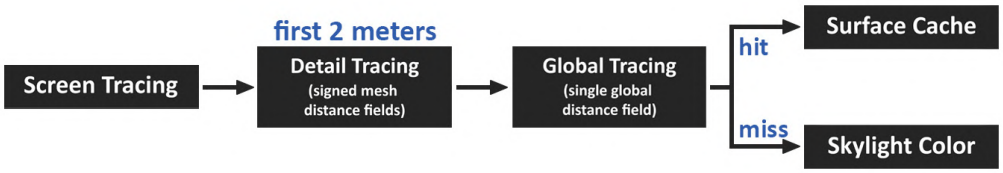



**Figure 4.5:** A scheme of Lumen's software ray tracing pipeline.

In this Section, I will describe the most relevant aspects of the distance field representation Lumen uses, along with notes on performance and its advantages and disadvantages when compared to the more traditional approach used by its hardware counterpart.

### 4.5.1 Signed Distance Fields

*Distance field (DF)* is generally a function that, for a given point $x$, returns a positive real value indicating the shortest distance of that point to a surface defined by this function. A slightly more advanced version of this concept, called a *signed distance field (SDF)*, also returns a positive or negative sign. This allows us to determine whether $x$ lies outside the surface's boundary or inside it, in which case the sign is negative. This is useful for *sphere tracing*, the most commonly used algorithm to find intersections with distance fields, described in Section 4.5.3 [Sum23].

A mathematical description of an SDF is shown below.

$$SDF(x) = \begin{cases} dist(x, \partial\Omega) & \text{if } x \in \Omega \\ -dist(x, \partial\Omega) & \text{if } x \notin \Omega \end{cases} \tag{4.1}$$

Here, $\Omega$ denotes the space bounded by the object's surface and *dist* is its corresponding distance function. This is suitable for objects which can be expressed using simple equations, such as a sphere or a line segment (for specific examples, please refer to Quilez [Qui]). For more complicated meshes with hundreds or thousands of triangles, a pure mathematical representation is no longer suitable. Therefore, Unreal Engine (not Lumen specifically, as DFs are also used for other techniques, such as ambient occlusion) uses a discrete *mesh distance field (MDF)* representation instead [Sum23, Gamd].

### 4.5.2 Mesh Distance Fields

Mesh distance fields are represented using sparse virtual volumetric textures that encompass the space near the surface (narrow band). Each voxel stores a positive or negative distance to the mesh's nearest surface, similar to typical signed distance fields. The default voxel density of MDF is 0.2 multiplied by the object's scale. This can be further increased (or decreased) using a project-wide setting or on an individual mesh basis [WN23, Gamd, Unr21].

The base distance field is then used for generating other mip levels, where each subsequent mip has half the spatial resolution than the previous

one but double the maximum voxel distance from the surface. During the *Lumen Scene Update* call, a compute shader calculates the distance of each MDF to the camera and chooses its mip level accordingly. The mips are then streamed in and out of memory by the CPU. A visualization of three different mip levels for a single mesh distance field can be seen in Figure 4.6 [WN23, Gamd, Unr21].

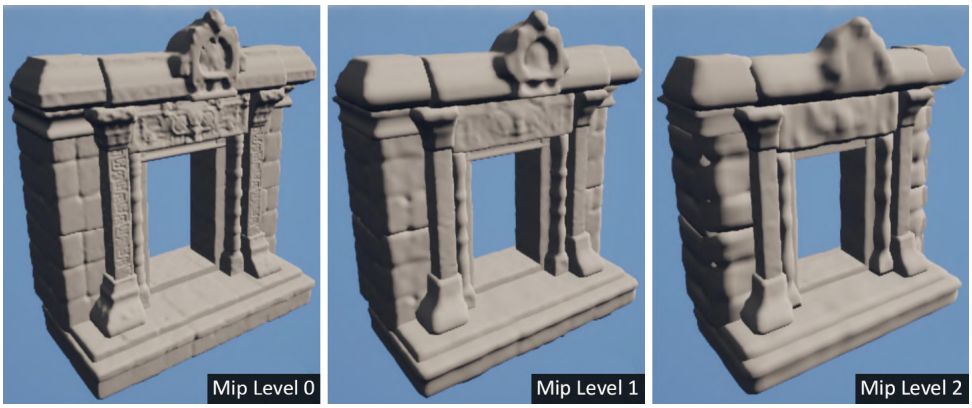



**Figure 4.6:** An showcase of three different mip levels of a mesh SDF (source: [WNK]).

To create an MDF from a given mesh, Unreal Engine utilizes *Embree*, which is a highly performant ray tracing library designed to run on the CPU. After defining the voxel grid based on density and object scale, for each voxel, Embree's function *rtcPointQuery* is used to find the shortest distance to the mesh's triangles. Afterwards, 64 uniformly sampled rays are cast from the center of the voxel. Based on the number of these rays which intersect a back face of the mesh, the method determines whether the voxel lies inside or outside its boundary, adjusting its sign accordingly [WNK].

There are a few edge-case meshes that need further handling, otherwise tracing against them could cause light leaking. For example, one sided surfaces have their negative space wrapped after 4 voxels. Similarly, meshes thinner than the default voxel size need to be expanded. However, this fix causes issues with over-occlusion and requires additional heuristics to work correctly with reflections. For more details, please refer to Wright [WNK].

### 4.5.3 Sphere Tracing

Tracing against distance fields is typically done using a method called *sphere tracing*, which is a subclass of *ray marching*. Ray marching generally works by iteratively moving along a ray until we find an intersection with an object. This can be done with constant distance steps, but that is usually too inefficient if the scene contains a lot of empty space. Sphere tracing (visualized in Figure 4.7) instead utilizes distance fields to skip the largest possible distance during each iteration [Har95].

To achieve this, we sample each relevant distance field to find the smallest possible signed distance from the current point on the ray. If we march along

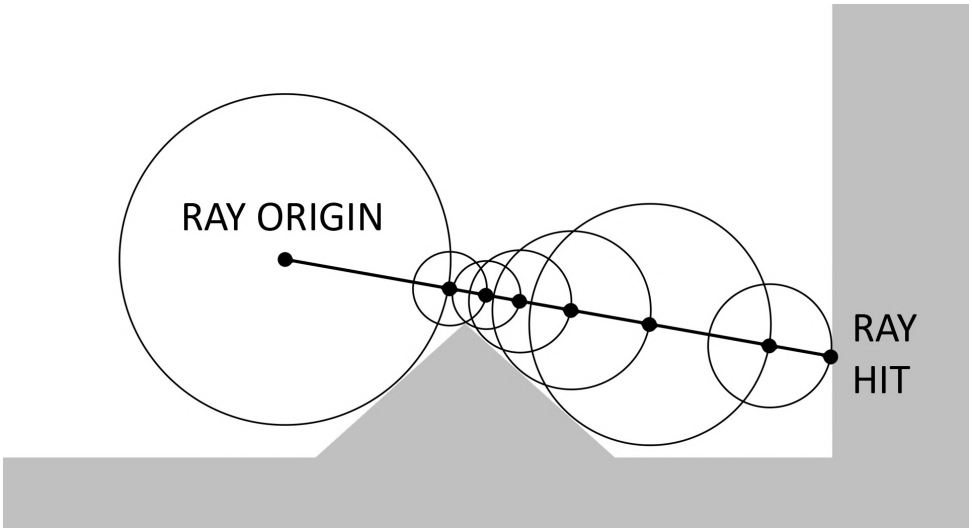



**Figure 4.7:** An illustration of the sphere tracing algorithm (source: [Tea22]).

the ray by this signed distance, we are guaranteed to either not hit anything or get so close to one of the surfaces that we can register a hit. This gives us a position in the world space where the intersection with an object occurred. When using mesh distance fields, this maximum is stored in the corresponding voxels and sampled from them. However, the number of iterations in Unreal Engine is limited to 64 for performance reasons [WNK, Har95].

Lumen also performs additional culling steps (before tracing begins), which are required to lower the number of sampled distance fields, thus accelerating the process. First, all objects outside the view frustum are culled. Second, for each *froxel* (frustum-aligned voxel) an array of MDFs intersect-able by rays going through the froxel from camera is created. Froxels without any geometry are skipped. During the tracing step, each ray is traced only against the MDFs in the array belonging to the froxel where the ray starts [WNK].

### 4.5.4 MDF Material Sampling

Since voxels do not contain any information about the UV or materials of the underlying geometry (and only provide the distance along with the corresponding mesh instance ID), calculating lighting at the hit point requires an additional material-holding data structure that can be easily sampled.

For Lumen, this is called *the surface cache*, and its implementation details are described in Section 4.7. Note that the surface cache still requires a normal vector to return a sample at the hit position. Generally, normal vectors can be estimated by evaluating the distance field's *gradient* at the hit position. In Unreal Engine, the gradient is numerically approximated using *central differencing* with 6 uniformly distributed samples around the hit position [WNK, Sum23, Unr21].

### 4.5.5 Global Distance Field

The *global signed distance field (GSDF)* is the main trick Lumen utilizes to speed up its software ray tracing. It is a dynamically updated structure that uses 4 *clipmaps* to represent the merged scene near the camera at different levels of detail. This allows for an extremely fast sphere tracing evaluation, as during each step, we no longer need to iterate through voxels of each MDF. Instead, we simply sample the nearest voxel of the GSDF and march along the ray accordingly. If the ray goes out of the clipmap's bounds, it simply switches to the next clipmap, which is less detailed [WNK, AA05].

To avoid introducing a significant overhead during updating, GSDF uses a static and dynamic cache. Similarly to other parts of Lumen (such as the surface cache), only the dynamic part of the scene which changed since the last frame is updated. This can sometimes be just a partial update, as the update budget for each frame is fixed, and MDFs in the more detailed clipmaps have a priority). [WNK, Sko23].

An example of a global signed distance field, along with its lit version from the Lumen scene, is shown in Figure 4.8.

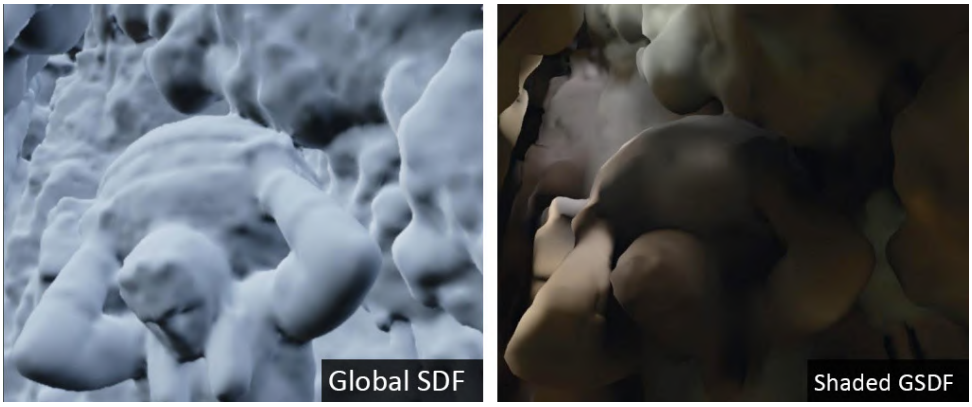



**Figure 4.8:** A visualization of the global signed distance field (source: [WNK]).

### 4.5.6 Height Maps

Since the highly detailed terrains created by Unreal Engine's *Landscape* tools are typically ever-present in most of the scenes, they require a dedicated representation for a higher tracing precision. This is done by dividing the landscape into equally sized components and storing them in memory as *height maps* (textures with one grayscale color channel, where each pixel represents the elevation at a given point).

During the tracing stage, a height map of the closest landscape component is marched through in transformed 2D (texture) space with constant steps. The goal is to find a pair of points where the height of the first one is below the height of the ray (zero) and the second above the ray. Then the hit point is approximated by linearly interpolating between the two obtained heights

and transforming it back to a 3D space. Its normal vector is perpendicular to the line segment that connects the two sampled points [WNK].

### 4.5.7 Advantages

As a result of being cleverly split into two phases and using other aforementioned optimization tricks, Wright claims that software ray tracing is slightly faster than its hardware counterpart (although this depends on the complexity and scale of the scene and the amount of overlapping objects). Software ray can be further accelerated by disabling the *detail tracing* in the project settings. This removes the step of tracing against the individual mesh distance fields from the pipeline [WN23, Gam23].

Software ray tracing also supports a wider variety of hardware as it runs on any GPU that supports Shader Model 5 (SM5) and DirectX11 [Gamc]. DXR support is also not required for software ray tracing, which makes it more widely available.

### 4.5.8 Disadvantages

As distance field representation is only an approximation of polygonal meshes, traces against it can sometimes cause light leaking, especially if the original geometry's dimensions are smaller than one voxel. The surface expand mentioned in Section 4.5.2 can help with that, but artifacts can still rarely occur. While this problem can be mitigated by setting the *Voxel Density* option to a higher value, it leads to a corresponding cubic increase in performance cost.

The visual imperfections for diffuse surfaces are usually covered by the screen traces, however, during my testing, I encountered a lot of mismatches between an object and its mirror reflection caused by the reliance on MDFs. This problem is shown in Section 5.2.5.

Another substantial issue with Lumen's software ray tracing is its inability to support dynamic meshes, i.e meshes with deformable geometry, as the mesh distance field representation is precalculated and supports only rigid transformations.

Finally, Lumen's specular reflections are also severely limited unless using hardware ray tracing, as without it, there is no support for multiple reflection bounces [Gamc, WNK].

## 4.6 Hardware Ray Tracing

Lumen's hardware pipeline (schematically shown in Figure 4.9) resembles the more traditional approaches to ray tracing, yet it still has its own tricks which give it a significant performance boost when compared to its predecessor from Unreal Engine 4 [WNK, Sko23].

While the MDFs used in the software pipeline are fast to trace against, they only approximate the underlying geometry and can thus produce incorrect results for detailed meshes. Hardware ray tracing on the other hand uses the

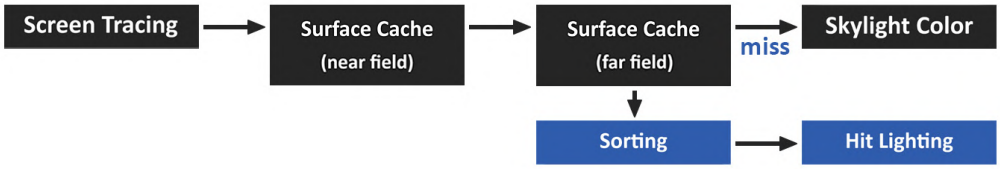



**Figure 4.9:** A scheme of Lumen's ray tracing when using hardware ray tracing. The steps highlighted in blue are performed only when using the hit-lighting pipeline.

full original geometry for short distances from camera and a reduced *far field* representation for larger distances. This allows for getting more perceptually accurate hit points while still maintaining a reasonable performance for larger scenes.

However, material and lighting values are still sampled from the surface cache, which gives a noticeable performance increase at the cost of limiting lighting to static meshes. This is called *the surface cache pipeline*. To mitigate these drawbacks, its extended version, called *the hit lighting pipeline*, can be used. Both of these approaches are described in the following Sections and it is possible to select which one of them Lumen uses via a project-wide setting.

Note that Lumen's hardware ray tracing uses many features of DXR, such as acceleration structures and special shaders. While the specific DXR implementation details are beyond the scope of this thesis, they are outlined in Wright's SIGGRAPH 2022 presentation [WNK] and mentioned in the following Section. For more information on DXR, please refer to its official documentation [Mic].

For a detailed comparison between software and hardware ray tracing, please refer to Sections 4.3.1, 4.5.7, 4.5.8 and 4.2. The most important disadvantage of hardware ray tracing I will highlight here is its vulnerability to overlapping objects. This is caused by the BVH used as an acceleration structure, which gives the biggest performance boost if all objects have a non-overlapping geometry and can therefore be efficiently culled. This issue is explained and illustrated in Section 3.4.2.

### 4.6.1 The Surface Cache Pipeline

The surface cache pipeline was introduced as a way to not only increase the performance of the previous UE4's *ray traced reflections* method, but also to fix its lack of specular occlusion mostly noticeable in mirror reflections.

To achieve a noticeable performance increase (and lower memory overhead), Lumen limits the traced geometry to static opaque surfaces. Using DXR terms, this means that the surface cache pipeline is limited to one *closest-hit shader* which fetches only the hit point and the normal, regardless of the object's material. As a consequence, the size of *payload* (a data structure storing information about the ray's path and intersected materials) is reduced to 20 bytes (from the original 64) [WNK].

As the lighting data stored in the surface cache converges to the full GI,

specular occlusion is inherently accounted for, which solves the second mentioned issue. This is shown in Figure 4.10 taken from the official SIGGRAPH presentation (2022). Notice the lack of direct lighting on the animated (non-static) character, which is not present in the image rendered using the *hit-lighting pipeline*.

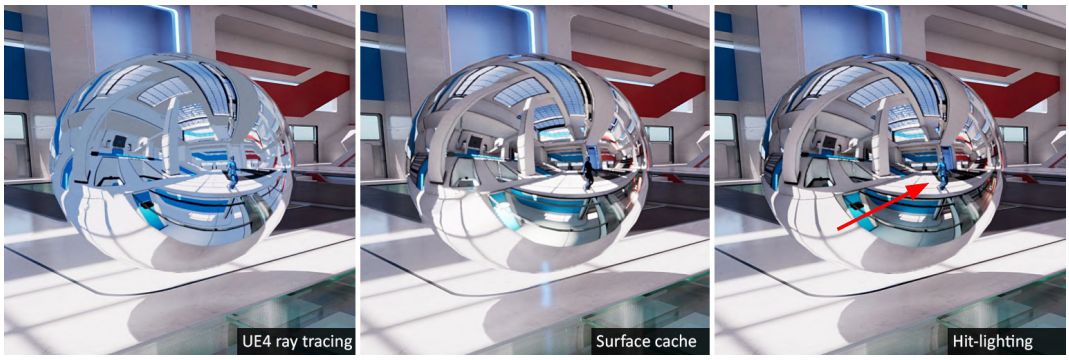



**Figure 4.10:** A comparison of Unreal Engine 4's ray traced reflections to Lumen's hardware ray tracing pipelines (source: [WNK]).

Note that apart from the aforementioned shader and payload changes, the surface cache pipeline also utilizes some features of DXR 1.1, namely *indirect ray dispatch* and *in-line tracing*. This further improves performance and flexibility by limiting the amount of necessary communication between the CPU and the GPU [WNK, Mic].

Transmissive materials are entirely skipped per ray based on the *MaxTranslucentSkipCount* parameter, which can be modified using an in-editor console command. Decreasing its value can yield better performance but will produce visually incorrect results, as unskipped objects will simply appear black [WNK].

### 4.6.2 The Hit-Lighting Pipeline

The hit lighting-pipeline enhances hardware ray tracing by using surface cache samples only for the indirect lighting. Materials and direct lighting are obtained separately, with a more precise secondary ray tracing. Apart from sharper visual quality, hit-lighting also enables direct lighting on dynamic meshes in reflections (though during my testing, it seemed this was not the case) [WNK].

The specific way in which the hit-lighting pipeline differs is that after the surface cache step, all rays are sorted by hit material IDs to improve SIMD efficiency. Afterwards, they are queued for an additional step, where they are evaluated using a variation of the Whitted ray tracing (mentioned in Section 3.3.1). More details about the sorting step and its purpose are described (and related to Fortnite) in Chapter 48 of *Ray Tracing Gems II* [MSW21].

### 4.6.3 Far Fields

To accelerate hardware ray tracing, both of the aforementioned pipelines use *hierarchical level of detail (HLOD)* to represent the geometry which is further from the camera than a given threshold (the default culling distance is 200 meters) [WNK].

Tracing against this *far-field* representation (which both simplifies the original meshes and merges them to clusters) is, as is typical for Lumen, handled in a different step than the so-called *near-field* tracing. The merged far-field geometry is then faster to trace against, as it contains fewer triangles and eliminates any overlap, making it more suitable for BVH. Both of these representations are shown in Figure 4.11 [WNK, Gamf].

**Figure 4.11:** A visualization of the different scene representations used in HW ray tracing. Near-field is shown in green, far-field in cyan, and the brown areas have missing surface cache coverage (source: [WNK]).

However, because of the low-level technical limitations of the pipeline, both the near-field and the far-field representations must be present in the BVH at the same times, bringing back the undesired object overlap. According to testing performed by Epic Games [WNK], the problem with these overlaps was so substantial that the performance suffered massively even when using ray masks to trace only against the currently active representation.

To remove these overlaps and fix the performance issues, the world position of the far-field geometry has a large global offset (which is then accounted for during the tracing) [WNK].

### 4.6.4 Supported Hardware

Lumen's hardware ray tracing currently supports the following platforms: [Gamc]

- PlayStation 5, Xbox Series S and Xbox Series X

- Computers with Windows 10 build 1909.1350 and newer, DirectX 12 (and DXR) support and any of the following GPUs:
  - NVIDIA RTX-2000 series or newer
  - AMD RX-6000 series or newer
  - Intel® Arc™ A-Series Graphics Cards or newer

It is expected that upcoming generations of both AMD and NVIDIA graphic cards will be supported as well, along with future Next-Gen consoles.

## 4.7 The Surface Cache

*The surface cache* is arguably the most crucial and innovative part of Lumen. It serves two primary purposes:

1. *MDF ray hit evaluation* — Tracing against the mesh signed distance field representation used in software ray tracing can only provide the hit point, normal vector, and mesh instance ID. To sample material and lighting values, a different surface representation is needed. This limitation is discussed in greater detail in Section 4.5.4 [Gamc, WNK, Sko23]. Note that, together with the distance field representation, the surface cache forms a so-called *Lumen Scene*, which can be visualized in Unreal Engine's editor. This visualization is particularly useful for identifying potential problems with geometry miss-matches or missing surface cache coverage [Gamc, Unr21].

2. *Caching of expensive calculations* — Computing high-quality, recursive multi-bounce ray tracing, which is essential for global illumination, is too slow for real-time applications. The surface cache addresses this by storing the calculated radiance values and slowly updating and propagating them (to itself) using a fixed time budget for each frame. In practice, this means that lighting and material changes are accumulated over multiple frames. The indirect diffuse lighting gather performed in the surface cache domain is internally referred to as *radiosity*, as it shares some characteristics with the technique described in Section 3.5 [WNK, Sko23, Unr21].

In this Section, I will provide a high-level overview of how the surface cache is represented in memory, as well as how Lumen generates it and updates it using *surfel cards*. Afterwards, I will briefly mention Nanite and its significance for dynamic surface cache updates. Finally, I will explain how the texture-space material sampling and light accumulation work. A few drawbacks and shortcomings of using surface cache will also be mentioned here and taken into account later during my testing of Lumen's performance and visual quality.

### 4.7.1 Texture Atlases

To start, the surface cache stores information about the scene near camera in nine block-compressed virtual texture atlases, each with a resolution of 4096x4096 texels (4k). The first five atlases represent the cached material and geometry properties (albedo, opacity, depth, normal, and emissive values). The next two cache direct and indirect lighting for the current frame, respectively. The eighth contains the accumulated radiosity over a given number of frames, and the ninth atlas stores the total radiance, which is then used in the *final gather* (described in Section 4.8). An example of the albedo atlas is shown in Figure 4.12. Each atlas is divided into 1024 physical *pages* composed of 128x128 texels, which are used for lookup and texel selection for dynamic updates [WNK, Unr21].

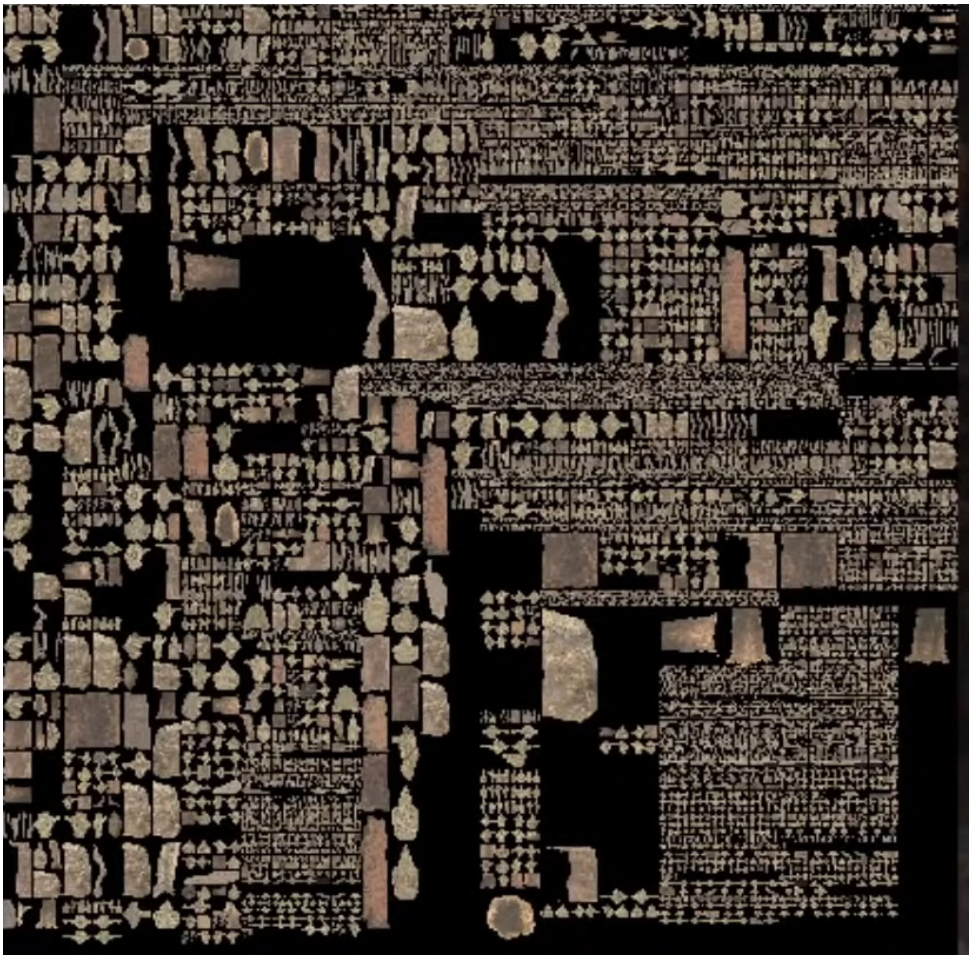


**Figure 4.12:** An example of a surface cache albedo atlas used in Epic Games' internal testing, presumably from the *Lumen in the Land of Nanite* tech demo (source: [Unr21]).

### 4.7.2 Cards

To differentiate between the represented object instances, Lumen uses pre-generated *cards*, which are uniform rectangular clusters of *surfels (surface elements)* aligned with the local coordinate axes of the corresponding object. During rendering, these cards are used for fast rasterization captures of the underlying geometry using orthographic projections. The results of these projections are then stored in the surface cache. The reason for evaluating and updating materials dynamically is the ability to cache distant meshes at lower resolution (or cull them entirely) to fit within the fixed memory budget and to support dynamic materials [WNK, Unr21].

The default material update budget for a single frame is 512x512 texels, which is only 1/64 of the atlas. The update order depends on the distance from the camera (closer cards are updated more frequently) and the last time the card was updated (to enable dynamic material changes for far-away objects) [WNK].

An example of card placement on a Fortnite asset is shown in Figure 4.13. This visualization can be seen for any mesh in the editor by using the *r.Lumen.Visualize.CardPlacement 1* console command [WNK, Unr21].

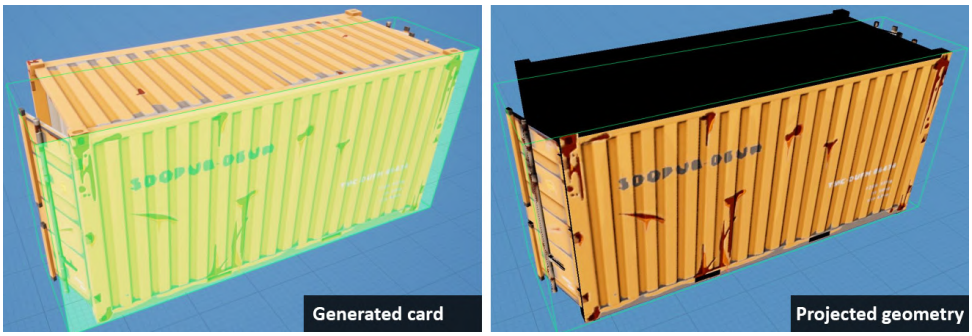



**Figure 4.13:** A visualization of a surface card (highlighted in semi-transparent green), along with its corresponding surfels (enclosed within the green bounding box), which will be projected into the surface cache during runtime. [WNK]).

By default, each mesh is covered by a maximum of 12 of these cards, though this limit can be individually raised in the editor to up to 40. This might be necessary for more complex objects, such as room walls in interior architecture visualization. Despite the existence of this option, Unreal Engine's official documentation [Gamc] recommends splitting complicated objects into separate meshes to prevent areas without surface cache coverage from appearing. An example of a mesh with missing card coverage is shown in Figure 4.14.

#### Card Generation

The offline process of generating cards is divided into three phases. First, the mesh is voxelized, and axis-aligned surfels are created. Afterwards, these surfels are grouped into clusters using a variation of the *K-means clustering*

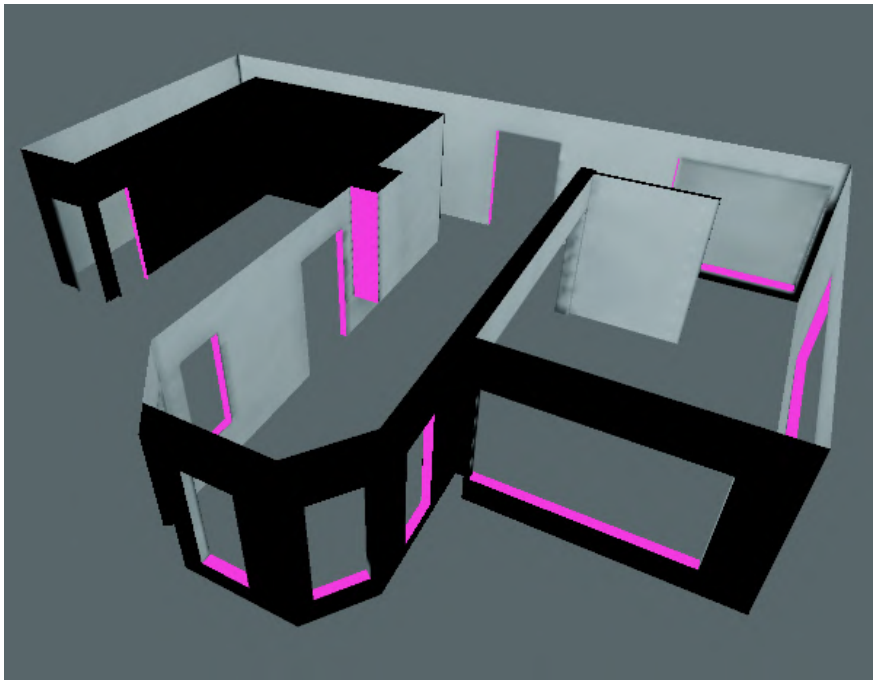


**Figure 4.14:** A problem with missing surface cache coverage. Parts of the mesh without corresponding cards are highlighted in pink. Back faces are shown in black (source: [Gamc]).

*algorithm*. Finally, the clusters are optimized through iterative parallel cluster regrowth and culling. If any of these steps fail due to undesirable properties of the geometry (such as a small size), a basic *cube map* is used to wrap the mesh [WNK].

The entire process is schematically shown in Figure 4.15 and its individual phases are further explained below. For a complete visualization of these steps, please refer to Wright [WNK].


**Figure 4.15:** A scheme of Lumen's card generation process (source: [WNK]).

1. *Surfel Coverage* — The mesh is uniformly divided (voxelized) into 3D cells with rectangular faces (2D cells). From each 2D cell, 64 rays are orthographically traced against the geometry. If a sufficient number of them hit the surface, a surfel is created.

   Following this, 64 additional uniformly distributed rays are traced for each surfel from the hemisphere around its normal. If the number of back-face hits is greater than the number of front-face hits, the surfel is marked as being positioned within the inner side of the surface and is therefore discarded. The surfel's occlusion, which is used in the next

step, is also determined from the average distance to the hits [WNK].

2. *Surfel Clusters* — As previously mentioned, the cluster generation is similar to K-means clustering (for more information on this algorithm, refer to Sharma [Sha19]). First, a random unassigned surfel is selected to be a centroid, and other surfels with matching orientation (normal vector) become candidates suited for joining its cluster. These candidates are then added to the cluster. Once none of them remain, a new centroid is calculated, and the cluster is regrown. This process repeats until the cluster stops changing or the iteration limit is reached.

   However, there a few differences to a typical K-mean clustering.

   First, new clusters are generated only after the previous cluster stops changing and not simultaneously based on the predetermined or given value of $K$ (indicating the number of clusters), as is usually the case. This means that instead of knowing the number of clusters beforehand, they are generated until each surfel belongs to one of the clusters or the iteration limit is reached.

   Second, candidates are weighted based on their distance to the cluster's bounds, their occlusion, and the effect their inclusion would have on the ratio of the cluster's side lengths. Square-like shapes are preferred.

3. *Cluster optimization* — In this phase, which might resemble K-mean clustering slightly more, the previously generated clusters are iteratively regrown in parallel. After each iteration, clusters containing fewer surfels than a certain threshold are culled, and new centroids are created in the resulting empty space. Once the iteration limit is reached, Lumen finally converts the $N$ largest clusters into cards, where $N$ is the number of cards specified in the object's settings.

A visualization of the previously shown object from Fortnite now fully covered in optimized surfel clusters can be seen in Figure 4.16.

### Nanite

As mentioned at the beginning of this thesis, Lumen is not the only innovation introduced in Unreal Engine 5. *Nanite* is the second and likely more prominent one, and while its detailed description is beyond the scope of this thesis, I find it important to at least mention this custom rasterization technique, since the performance of surface cache recapturing greatly benefits from its inclusion in Lumen's pipeline, as verified in Section 5.3.4 [WNK, Unr21, SIG21].

The core idea behind Nanite is that rendering performance should remain constant based on the number of pixels on our screen rather than scale linearly with the amount of geometry primitives in the scene, as is typically the case with conventional rasterization.

Nanite achieves this by converting the scene into a directed acyclic graph of triangle clusters, with dynamic, view-dependent LODs, where each mesh is represented in a highly compressed format. This allows for fast streaming

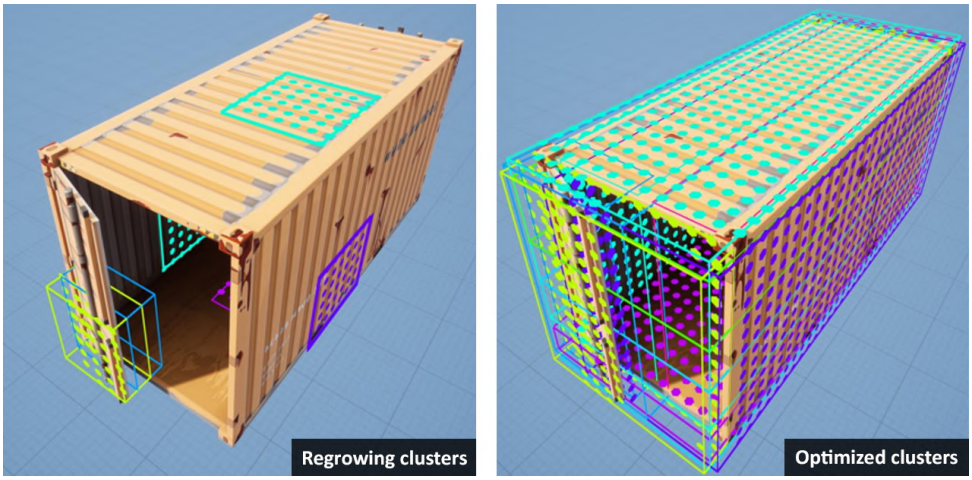



**Figure 4.16:** A visualization of the cluster regrowing process and an object fully covered in surfel clusters (source: [WNK]).

from SSD (or HDD), as branches at lower levels do not need to be stored in memory when they are not visible.

Overall, Nanite significantly reduces the number of draw calls in most scenarios. Another benefit is its automatic, view-dependent level of detail, which frees artists from the necessity to manually optimize high-poly meshes. An example of a scene filled with Nanite meshes composed of millions of triangles is shown in Figure 4.17.

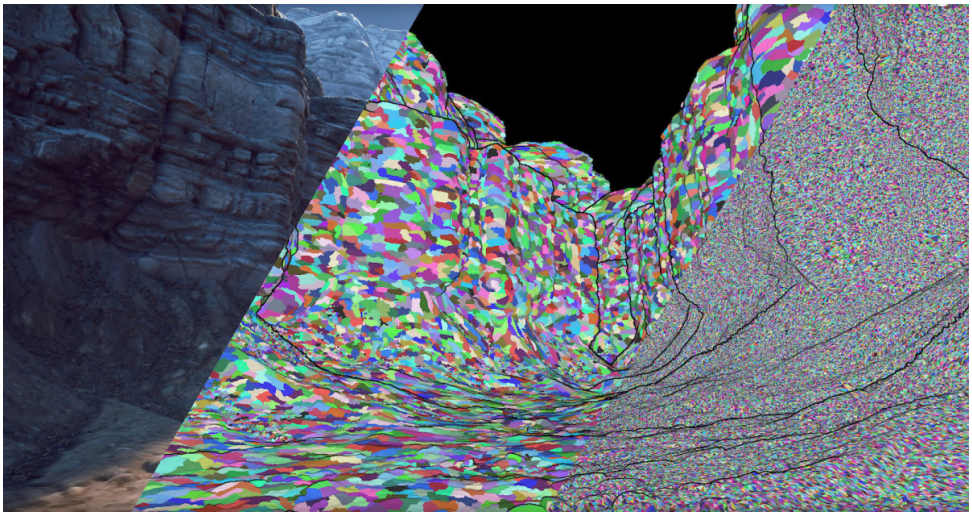


**Figure 4.17:** A comparison of a rendered scene and its visualized Nanite representation with two different levels of detail (source: [Ver21])

While using Nanite meshes for Lumen is not mandatory, it does make the surface cache card recapturing much faster, as it allows for producing multiple orthographic surface projections in parallel using the dedicated software rasterizer. Wright claims that according to their internal testing, surface cache updates are 10 to 100 times faster when using Nanite [Unr21],

particularly in more complex and detailed scenes. For this reason, some video games, such as Fortnite, do not even allow Lumen to be enabled without Nanite.

Relying on Nanite does have its drawbacks, however, especially when paired with hardware ray tracing, which cannot directly trace against Nanite geometry due to its dedicated data format. This limitation is further described in Section 4.4. For more details on Nanite, please refer to Juříček [Ju3], Nanite's documentation [Gam24b] or the official Deep Dive into Nanite from SIGGRAPH 2021 [SIG21].

Note that Nanite is enabled by default for each static mesh, automatically converting it into a so-called *Nanite mesh*. This setting can be toggled off in the mesh's properties or by using the context menu when selecting multiple meshes in the Unreal Engine's content browser [Gam24b].

### Card Merging

To simplify material recapturing, improve memory efficiency, and prevent large distant objects from being partially culled in the surface cache, cards are often merged at runtime based on their orientation and overlap. The merged groups are then captured using a simple cube map projection, which accelerates the update process, since the cards do not need to be projected to individually [WNK].

A simple example of such card merging is shown in Figure 4.18.

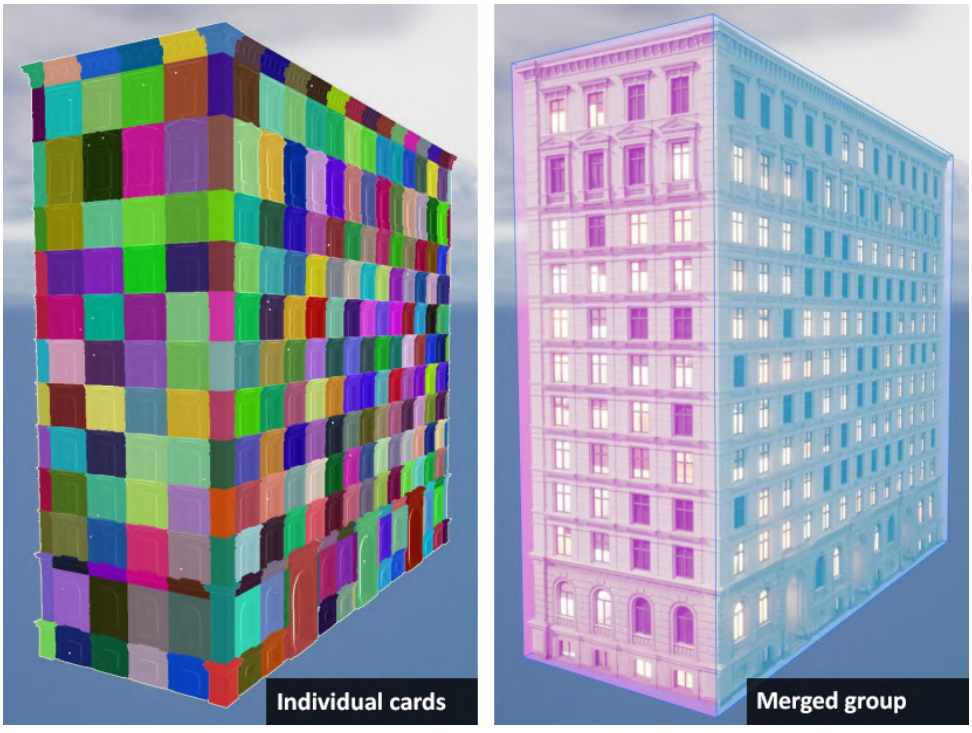



**Figure 4.18:** A comparison between a building covered by separate cards and its variant where all cards are merged into one (source: [WNK]).

### 4.7.3 Material Sampling

With card generation and material capturing explained, it is important to highlight how specifically a ray retrieves these cached material data (and radiance values) whenever a hit occurs. As mentioned in Sections 4.5 and 4.6, sampling the surface cache requires each ray to know a world-space hit position, a normal vector, and the mesh instance ID. This information is calculated in a shader on a per-ray basis.

Note that materials and lighting are sampled using separate functions, as the hit lighting hardware ray tracing pipeline utilizes surface cache only for indirect lighting. However, the process of selecting the correct cards and their corresponding coordinates in the material and lighting atlases remains the same across the whole pipeline and can be described as follows: [WNK]

1. Given a mesh instance ID, Lumen fetches the relevant data from a large global buffer and stores them in a *MeshCardsData* structure. This includes up to six best cards (one for each axis orientation: +x, -x, +y, -y, +z and -z) stored in a *CardLookup* array, a matrix used for world-to -local coordinate transformation, and optional bit tags which later modify the sampling bias, such as foliage and height field.

   Unfortunately, I was unable to determine how Lumen selects the six cards to store in the global structure. Initially, I believed that instead of individual cards, a list of equally oriented cards is stored at each index. However, both Daniel Wright's presentation from SIGGRAPH 2022 [WNK] and the source code indicate that this is not the case.

2. The hit point and normal vector are transformed into the local mesh space using the aforementioned transformation matrix. Then two tests are performed to cull the six previously selected cards to a maximum of three.

3. First, the transformed normal vector is squared, and for each of its resulting positive components, the card corresponding to that direction is added to a card bit mask. The reason for sampling multiple cards instead of one is related to non-axis-aligned faces, which need weighted samples from all of their orthographic projections in order to correctly reconstruct the original values. This is demonstrated in an example shown in Figure 4.19. For axis-aligned hit points, only one card is generally sampled.

4. Second, Lumen checks whether the transformed hit point lies within the bounding boxes of each of the remaining cards, using a specified bias. Each bounding box is represented by the card's center position and its *extent*, which is a 3D vector where each component corresponds to the distance from the center along its respective axis.

5. All remaining cards that were not masked out are sampled using a dedicated function called *SampleLumenCard*. This function converts the

hit position and normal vector from local mesh space to the specific card space, calculates the nearest atlas UV coordinates of the hit point, and checks the sample's validity and visibility. A *normal weight* is calculated based on the alignment between the hit normal vector and the normal vector of the card. This prevents potential stretching artifacts caused by the orthographic projection.

6. Using the computed UV coordinates, four nearest depth values are gathered from the depth atlas, and a normalized hit distance from the card's center is calculated. These depths are then compared with this normalized hit distance to approximate the occlusion of each texel, which is used as its secondary weight. Finally, all four *texels weights* are bilinearly filtered, multiplied by their visibility and the normal weight, and used for sampling from the other material and lighting atlases.

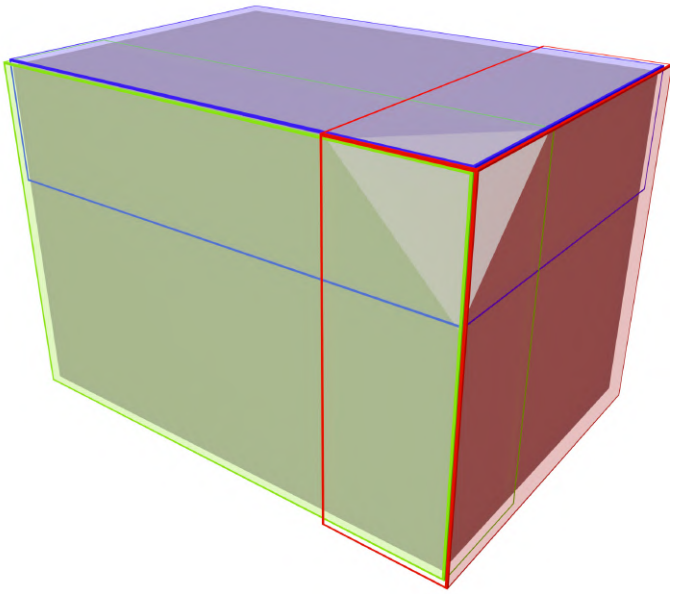


**Figure 4.19:** A visualization of a simple mesh covered with surfel cards. Any surface point inside the corner triangular face is sampled using all three cards, as it lies within all their bounding boxes. This is indicated by the fact that all components of its squared normal vector have positive values. To retrieve correct cached material and lighting colors, the results of all three cards need to be weighted and blended together to avoid a stretched perspective.

For more implementation details on surface cache sampling, please refer to Unreal Engine's source code, specifically to the following filepath: *UnrealEngine / Engine / Shaders / Private / Lumen / SurfaceCache / LumenSurfaceCacheSampling.ush*.

### 4.7.4 Lighting Evaluation

Similarly to material values, lighting in the surface cache domain is updated each frame using a fixed budget, which is set by default to 1024x1024 texels for direct lighting and 512x512 texels for indirect lighting. This corresponds to 64 and 16 atlas pages, respectively. Direct lighting has a larger memory budget, as its changes affect the scene more prominently and it is faster to calculate [WNK, Unr21]. Note that these update budgets can be changed using the *Lumen Scene Lighting Update Speed* setting, which improves propagation speed of both direct and indirect lighting changes [WNK].

To propagate these changes as effectively as possible, Lumen keeps track of when each page was last used (either for sampling or value propagation) and updated. The page priority is then calculated as $Priority = LastUpdated - LastUsed$. Afterwards, Lumen builds a histogram containing the priority values, and pages with the highest priority are retrieved and updated until reaching the specified budgets for the given frame [WNK].

The update methods for both direct and indirect lighting are schematically shown in Figure 4.20 and conceptually described below.

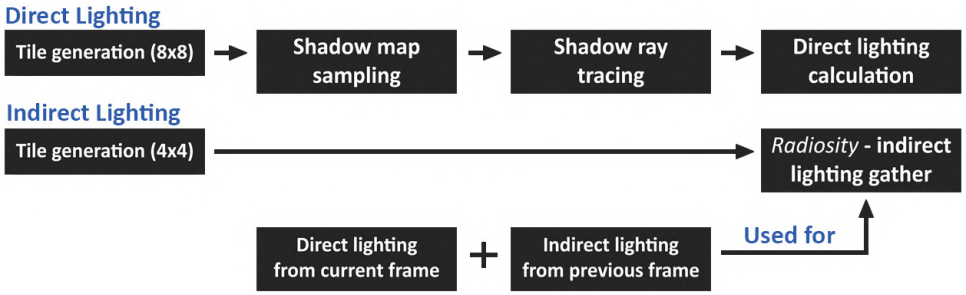



**Figure 4.20:** A scheme showing Lumen's surface cache lighting update methods.

#### Direct Lighting

To compute direct lighting, each atlas page is divided into 256 8x8 texel tiles. These tiles are then sorted based on their distance from the camera to improve coherence. Afterwards, for each tile, Lumen picks the first 8 shadow casting lights from the global array and tries to sample their shadow values from the available precalculated shadow maps to create an 8-bit shadow mask. If a shadow map sample is missing for any of these lights, the tile spawns a shadow ray [WNK].

These rays are compacted and in the next step, they are all traced to find an occluder between the tile and the remaining light sources to mark their corresponding bits in the shadow mask. Once the shadow mask is completed, direct lighting contribution is calculated for all light sources visible from the tile. Then for each texel in that tile, this incoming radiance is stored in the direct lighting atlas [WNK].

### Indirect Lighting

The indirect lighting gather is essentially Lumen's *final gather* performed in a texture space. I describe it in greater detail in Section 4.8.5, but its core idea is to trace rays from probes placed on 4x4 tiles belonging to the selected pages. The sampled radiance, obtained from the current direct lighting atlas and the indirect lighting atlas from the previous frame, is interpolated in the heavily downsampled probe space and stored in the current frame's indirect lighting atlas. The reliance on previously calculated values ensures that, given a stable static scene, the lighting slowly but steadily converges to infinite diffuse bounces, much like *radiosity* (described in Section 3.5) with enough iterations [WNK].

## 4.8 The Final Gather

Once the surface cache is updated, Lumen can finally gather the cached data and use them to light the individual screen pixels according to the Monte Carlo approximation of the rendering equation (described in Sections 2.4.4 and 2.4.5).

However, as mentioned in the previous Chapter, stochastic ray tracing algorithms (such as path tracing) are very resource-intensive for real time global illumination since the number of rays cast per pixel typically needs to be in hundreds to achieve a noise-free image. According to Wright [Wri21], Lumen has a budget of only around a half ray per pixel to achieve real-time framerates on next-gen consoles, which is significantly lower than the required number. To be more specific, Unreal Engine 5's offline path tracer requires an average of 100 samples for outdoor scenes. Indoor scenes, which are often lit mainly by indirect lighting, require around 500 samples [Wri21, Unr21].

To overcome this issue, Lumen's *final gather* takes advantage of the fact that incoming radiance at any surface point $x$ is mostly spatially and temporally coherent [Kri, Wri21], thus instead of tracing rays from each pixel of the screen, it strategically places probes on the depth buffer, projects them to the geometry, and traces from them in world space. The incoming radiance is heavily downsampled and spatially filtered in probe space, but the integration with BRDF is performed at full resolution to ensure that no geometrical detail is lost.

This method, called *screen space radiance caching (SSRC)*, combined with sampling cached lighting from the surface cache and utilizing *world space radiance cache (WSRC)* for more stable distant lighting, allows Lumen to approximate the full diffuse path $LD^*E$ in real time with its limited budget of half a ray per pixel. Glossy and specular reflections below a certain roughness threshold are handled separately, which is described in Section 4.9 [WNK, Wri21, Sko23].

The full final gather is schematically shown in Figure 4.21, and its individual components are described in the following Subsections. I was mainly focused on the SSRC, as it is the most innovative technique, but the

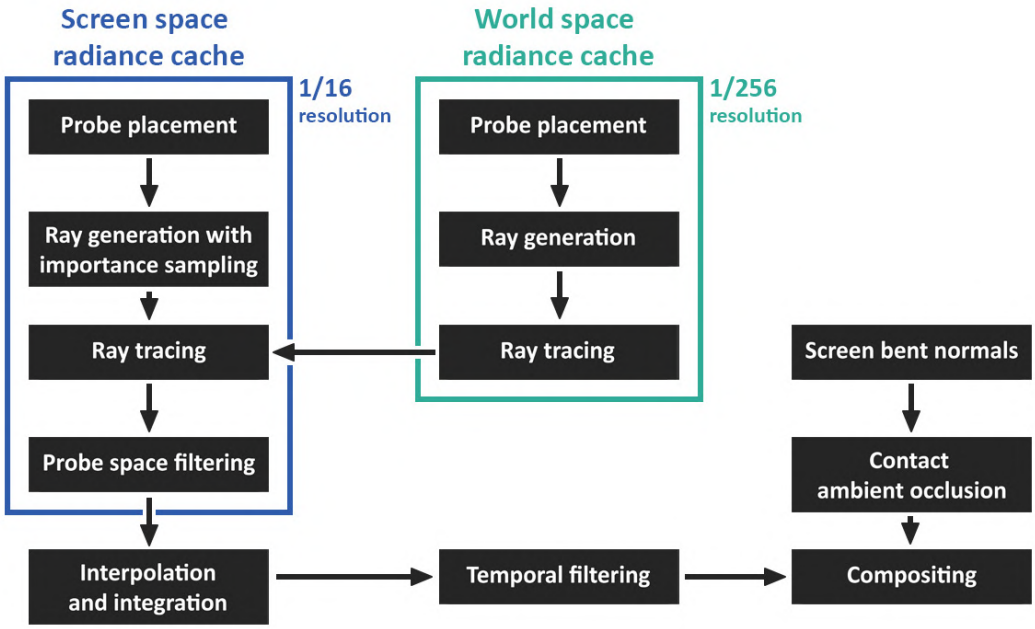



**Figure 4.21:** A scheme showing Lumen's final gather. Note that all its steps except for radiance caching are performed at full resolution.

other steps are also mentioned for completeness. If you wish to learn more official information about any part of the final gather, please refer to Daniel Wright's talk from SIGGRAPH 2021 [Wri21].

### 4.8.1 Screen Space Radiance Cache

As already mentioned, the core idea of screen space radiance caching is to place a uniform (but adaptive) grid of probes directly on the z-buffer and sample the radiance from a fixed number of incoming directions directly from the surface cache. The probe placement process, called *hierarchical refinement* with *adaptive super-sampling*, is described in the following subsection [Wri21].

Each probe can be imagined as a uniform octahedron, where, for each of its sides, Lumen casts 8 rays with slightly jittered directions. This means that for each probe, 64 traces are performed to sample the incoming radiance from the surface cache. Note that the surface cache regularly updates its accumulated direct and indirect lighting, albeit with limited quality and precision.

In memory, each probe is represented as an 8x8 array of texels. As such, all cached radiance values for all probes are stored in a large texture atlas, which is shown in Figure 4.22. To simplify lookup, filtering, and the final gather's *structured product importance sampling* (described in Section 4.8.1), all probes are indexed based on their position. Similarly, all rays in a single probe are consistently indexed and have matching directions with those in neighboring probes [Wri21, WNK].

Note that Lumen keeps track of the screen space radiance cache from

**Figure 4.22:** A visualization of the screen space radiance cache probe atlas. Note that all adaptively placed probes are stored at the bottom of the atlas to avoid the necessity of using another data structure and separate processing step (source: [Wri21]).

previous frames, which is used for temporal accumulation to reduce noise in the current frame. To increase performance, all the steps mentioned below (such as probe placement, importance sampling, ray generation, and filtering) are executed by the GPU using compute shaders with heavily optimized thread distribution [Wri21, WNK].

### Screen Probe Placement

The probe placement is an iterative process. First, probes are uniformly placed on every 16th pixel in both screen directions. When multiplied by the aforementioned 64 traces per probe, this would set the final gather's cost at 1/4 of a ray per pixel. However, in areas with very detailed geometry, having a grid this spatially sparse can cause interpolation (described in Section 4.8.3) to fail for some pixels, since no probes may lie within their depth plane or close to it. This issue is illustrated in Figure 4.23 [WNK, Wri21].

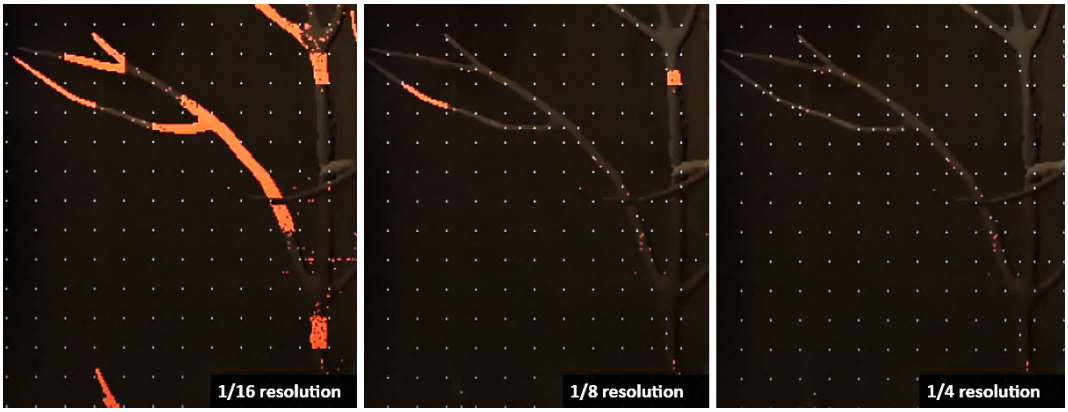



**Figure 4.23:** A visualization of Lumen's hierarchical refinement for probe placement, where every white dot represents a single probe from the screen space radiance cache. At each level, the pixels with failed interpolation are highlighted in red (source: [Wri21]).

### Structured Product Importance Sampling

Stochastic ray tracing methods typically rely on *importance sampling* to ensure that no computation time is wasted on samples that provide little no value to the final result, thereby decreasing variance in the color intensity of nearby pixels (noise). Lumen's final gather operates in a downsampled space, and thus the frequency of its noise is significantly lower, making it less noticeable (see Figures 4.25 and 4.26). However, its presence still worsens the visual quality of the resulting image, especially since it is temporarily unstable.

For this reason, Lumen also relies on importance sampling to reduce noise even before the spatial and temporal filtering steps. I described the core idea of importance sampling in Section 3.4.1, where I introduced the following Monte Carlo approximation of the illumination integral:

$$\lim_{N\to\infty} \frac{1}{N} \sum_{i=1}^{N} \frac{L_i(x,\omega_\text{i}) BRDF_r(x,\omega_\text{o},\omega_\text{i}) \cos\theta_\text{i}}{PDF(\omega_\text{i})} \tag{4.2}$$

Typically, simple ray tracing algorithms bias only towards samples with higher $BRDF$ or cosine values, as the incoming radiance $L_i$ is what the rendering equation is trying to solve in the first place. Lumen's final gather uses a slightly more sophisticated method called *structured product importance sampling* [WNK].

As its name implies, Lumen estimates the PDF by using the entire illumination product $L_i(x,\omega_\text{i}) BRDF_r(x,\omega_\text{o},\omega_\text{i}) \cos\theta_\text{i}$, including the incoming radiance, which is estimated by values stored in the radiance cache from the previous frame. Since all probes and their rays are indexed, this reprojection is very fast. However, in cases where it fails (for example, due to the fast movement of the camera) a value from the more stable *world space radiance cache* is used. The $BRDF$ is accumulated and averaged from pixels that

will use the given probe for interpolation [Wri21]. All BRDF information is stored in the rasterized z-buffer.

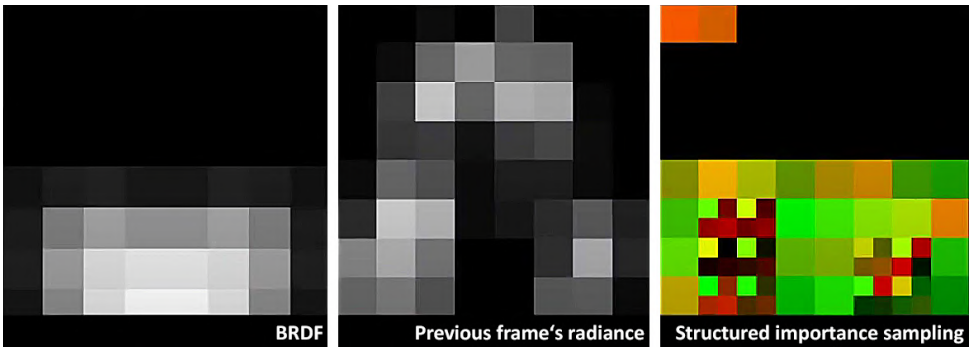



**Figure 4.24:** A visualization of structured product importance sampling in the texture domain. Most rays stored at the top are culled due to low BRDF and some of the rays with large BRDF and lighting product are super-sampled (source: [Wri21]).

The *structured* part of the name comes from the fact that this approach is inspired by Agarwal's article called *Structured importance sampling of environment maps* [ARBJ03], which employs a hierarchical sampling structure.

Instead of altering directions of all cast rays (as previously mentioned, their distribution is uniform), Lumen super-samples the most influential rays. Specifically, each probe maintains a mip quad-tree variant of the stored radiance, where the rays culled by BRDF are redirected towards directions with the highest value of the illumination product, with a small random offset. Note that incoming radiance from previous frame is not used for culling, as it is susceptible to variance, unlike BRDF [Wri21].

After the ray tracing is completed, the radiance values obtained from the super-sampled directions in the mip variant are averaged and composed into the original radiance cache. This structured importance sampling process can be seen in the probe texture space in Figure 4.24, while its impact on noise reduction is illustrated in Figure 4.25 [Wri21].

### Probe Space Filtering

The last operation Lumen's final gather performs in downsampled space is spatial filtering for the purposes of further reducing noise and smoothing out the cached radiance data. To achieve this, each probe loops over its four nearest neighbors (horizontal and vertical) and averages their weighted values. The list of filtered neighbors can be optionally extended to diagonals and second nearest probes. This extension is used in conjunction with lowering the strength of the temporal filter in order to remove flickering artifacts caused by fast-moving objects [WNK, Wri21].

The weights are determined by the depth and angular differences of the filtered probe and its neighbors. More specifically, a single position weight is calculated using the following exponential function:


**Figure 4.25:** A comparison of a scene rendered using Lumen with uniform (left) and structured product importance sampling (right). The rays originating from one screen probe are visualized, with the super-sampled ones highlighted in white. Notice the substantially lower amount of noise in the right image, especially in the corners of the room (source: [WNK]).

$$\text{PositionWeight} = e^{-\text{SpatialFilterPositionWeightScale} \times (\text{RelativeDepthDifference})^2} \quad (4.3)$$

Here, $RelativeDepthDifference$ represents a normalized depth difference between the position of the current probe and its neighbor. The other variable is a constant that can be modified through a console command in Unreal Engine's editor, with a default value of 1000.

Using an exponential function ensures that, with increasing depth differences, the weights have substantially lower values, which helps to reduce light leaking artifacts [Wri21].

$$\text{AngleWeight} = 1.0 - \text{clamp}\left(\frac{\text{NeighborAngle}}{\text{SpatialFilterMaxRadianceHitAngle}}\right) \quad (4.4)$$

The angle weight is calculated by first clamping the ray distance of the neighbor probe to the hit distance of the corresponding ray in the current probe. Next, the vector between this position and the probe location is calculated, and the angle between this newly created ray and the original one is measured. Afterwards, this angle is divided by a constant $SpatialFilterMaxRadianceHitAngle$ (with a default value of $x$), clamped to the range $[0, 1]$ and subtracted from one. This ensures that filtering does not cause excessive light leaking, since directions where $NeighborAngle$ is too high receive a much lower weight.

Finally, the position (depth) and angle weights are multiplied together and used to weight the incoming neighbor radiance value.

An example of Lumen's spatial filtering is shown in Figure 4.26. A more intuitive explanation of this approach, along with an illustration of the angle weighting, can be found in Daniel Wright's video on radiance caching [Wri21].


**Figure 4.26:** A comparison of an image using an unfiltered and filtered radiance cache. Notice the substantially lower amount of noise in the right image (source: [Wri21]).

Notice the occasional lack of contact shadows caused by filtering in the downsampled space. This issue is addressed by compositing in ambient occlusion later using *bent normals*, which represent directional occlusion and are calculated in screen space at full resolution. The specific approach used by Lumen's final gather originates from the paper *Horizon-Based Indirect Lighting (HBL)* by Benoît Mayaux [May18], which is publicly available on GitHub [Wri21].

### 4.8.2 World Space Radiance Cache

In addition to the screen space radiance cache, Lumen also utilizes a sparse, clip-mapped, uniformly distributed volumetric grid of probes placed in world space near the screen probes, referred to as *the world space radiance cache (WSRC)* [Wri21].

The memory representation of this world space radiance cache is nearly identical to the screen radiance cache, except for the fact that its probes have a much higher directional resolution. Specifically, each probe traces 32 rays in 32 different directions for a total of 1024 traces. This makes WSRC more suitable for sampling distant light sources, which could otherwise be missed by the screen probes, as those have a lower directional resolution.

In practice, Lumen limits the ray distance for screen probes to 2 meters. Any radiance for hits further than that is obtained by interpolating the value of the best matching ray from the nearest world probe (meaning that WSRC ray tracing runs first).

Unlike screen probes, world probes have fixed positions near the scene's geometry, which are not temporarily jittered. This, combined with the assumed stability of distant lighting, allows all visible probes to persist across frames. To propagate lighting changes, a small fixed subset of them is retraced each frame. New probes can be allocated and deallocated only when the

camera or objects in the scene move. One such placement of the WSRC is shown in Figure 4.27. My test scenes created in Unreal Engine 5.5 also allow for showing its visualization through the user interface, which I recommend to anyone who is interested in seeing it dynamically adjust as the scene or its lighting change.

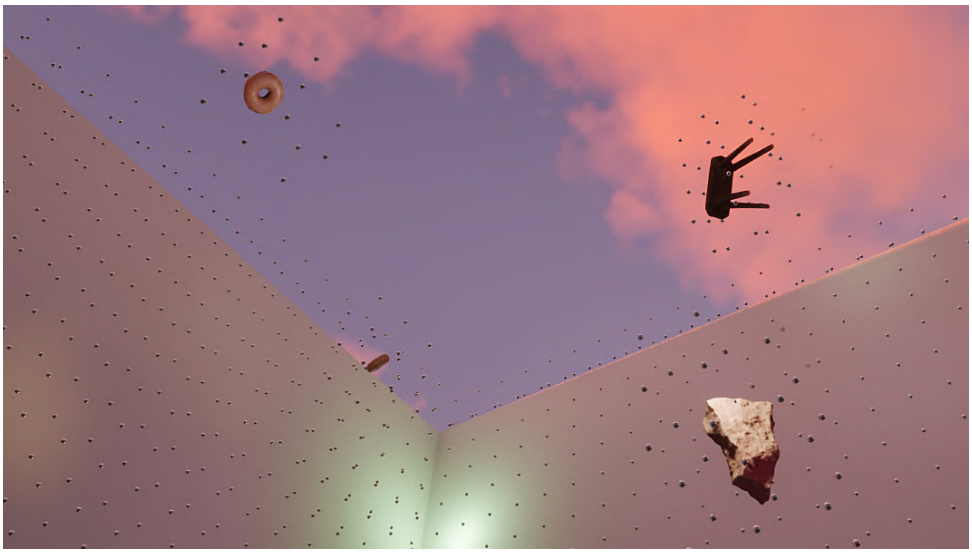


**Figure 4.27:** A visualization of the world space radiance cache used for distant lighting. Notice the sparse allocation around visible objects.

However, the dynamic adjustment of the WSRC probes means that the number of visible world probes can vary significantly between frames, which makes WSRC less suitable for product importance sampling. For context, the screen space radiance cache uses interpolated data from the world space radiance cache if the previous frame's cached radiance reprojection fails, but the world space radiance cache would lack such fallback mechanism if it tried to reuse the cached radiance values from the previous frame and failed [Wri21].

Therefore, the world space radiance cache relies only on BRDF for its importance sampling. Furthermore, probes near the camera have their directional resolution super-sampled to 64x64 texels (4096 traces) to ensure that nearly all important distant lighting is captured [WNK, Wri21].

During filtering, an occlusion check must be performed first, since no mutual visibility of neighbor probes is guaranteed [Wri21].

According to Wright, the inclusion of the world space radiance cache in Lumen's final gather greatly improves the temporal stability of the whole process [Wri21].

### 4.8.3 Interpolation and Integration

After spatial filtering, Lumen converts each octahedral probe into a *3rd order spherical harmonic (SH)* representation to improve load coherence for individual screen pixels [Wri21]. A detailed explanation of spherical harmonics

is well beyond the scope of this thesis, but on a high level, they are functions operating on a spherical domain. This makes them suitable for storing the incoming radiance, as it is a directional value. For more information on spherical harmonics and their relation to global illumination, please refer to the following article by Robert Green [Gre03].

After this conversion, the SH values are interpolated from probes to nearby screen pixels using a plane-distance weighting. Finally, the full Monte Carlo integration (described by equation 4.2) can be performed using these newly interpolated radiance values combined with full-resolution BRDF and normal vectors stored in rasterized buffers. This process is illustrated in Figure 4.28 [WNK, Wri21, Unr21, Sko23].

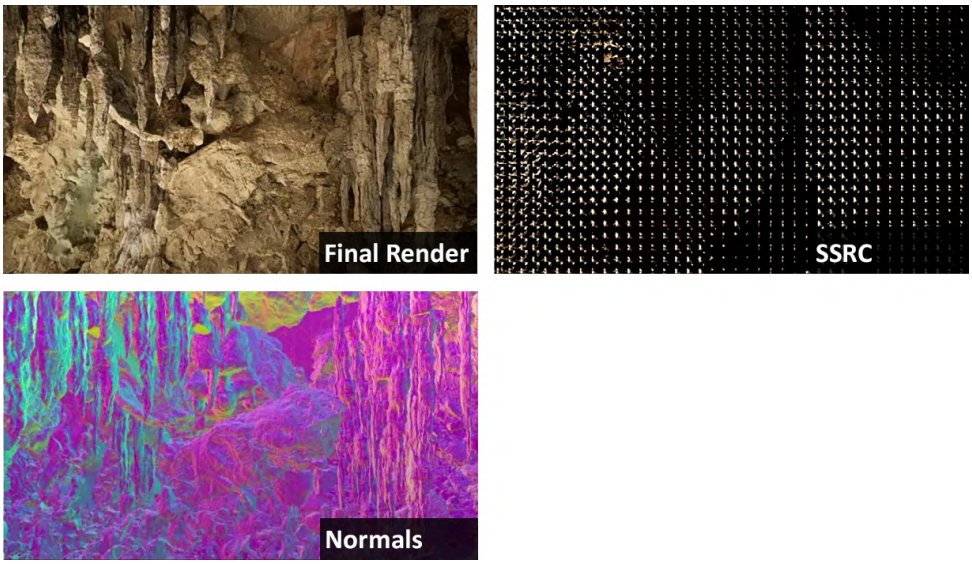



**Figure 4.28:** An example of a scene rendered using Lumen's final gather, along with its screen space radiance cache and full-resolution rasterized normals stored in a buffer (source: [Wri21]).

### 4.8.4 Temporal Filtering

As one of the final full-resolution steps, Lumen implements a temporal filter that reuses information from past frames to reduce artifacts caused by probe jittering. As is usually the case with temporal filtering, previously rendered pixels with substantial normal or depth difference to the one from current frame are weighted or entirely excluded from filtering. The inclusion of a temporal filter also further reduces noise in the final image [Wri21, WNK].

One thing which makes Lumen's filtering stand out is that its ray tracing detects fast moving objects. As previously hinted at in Section 4.8.1, when a fast moving object is detected, the influence of temporal filter is reduced and the space spatial filter gets an increased filtering radius. This prevents ghosting errors, which can manifest as slowly disappearing shadows behind dynamic objects [WNK].

After the temporal filter finishes, ambient occlusion is composited into the final image, and specular and glossy reflections are calculated, if enabled.

### 4.8.5 Other Domains

Adding to the complexity, Lumen's final gather has two specialized variants, both of which are described in great detail in Wright's presentation from SIGGRAPH 2022 [WNK]. In this section, I will cover their most important high-level differences from the previously described opaque final gather [Wri21, WNK].

#### Volumetric Final Gather

For volumetric and translucent objects, Lumen replaces the standard screen space radiance cache with a grid of 4x4 probes distributed uniformly within the view frustum. Depth test using HZB is performed for each probe to determine its visibility before ray tracing.

As is usually the case with Lumen, a lower-resolution variant is used for distant lighting. In this case, it is a yet another world space radiance cache (overlaid on top of the opaque one), where each probe has a directional resolution of 16x16 texels.

The ray tracing, integration, and filtering steps are mostly similar to the opaque gather. An example of a render that used Lumen's volumetric final gather is shown in Figure 4.29 [WNK].


**Figure 4.29:** A showcase of Lumen's volumetric final gather and its multiple steps (source: [WNK]).

#### The Surface Cache Final Gather

As hinted at in Section 4.7.4, Lumen uses a texture-space variant of the final gather to propagate indirect lighting throughout the surface cache. Surprisingly, only one level of probe distribution is used here. To be more specific, each 4x4 atlas tile in the surface cache is assigned to one 4x4 hemispherical probe. This placement is visualized in Figure 4.30 [WNK].

A subset of probes is selected every frame based on the indirect lighting update budget. Afterwards, 16 rays are traced for each of them. No

**Figure 4.30:** A showcase of probe placement in Lumen's final gather used for propagating indirect lighting within the surface cache (source: [WNK]).

importance sampling is utilized during ray generation, but the results are bilinearly interpolated for each texel from the 4 nearest probes. To avoid artifacts, Lumen uses depth map and plane order weighting to discard values obtained from these neighbor probes if they happen to be invisible from the texel due to occlusion.

The probe placement is also jittered based on the frame index, similarly to the opaque final gather [WNK].

## 4.9 Reflections

As already mentioned, Lumen's final gather only handles indirect reflections on opaque diffuse surfaces. To allow for light paths which contain multiple glossy or specular reflections, another optional technique, called Lumen Reflections, can be used afterwards.

Unreal Engine 5 uses *GGX* microfacet distribution [WMLT07] to control how rough, glossy, or specular a material is. This parameter, called roughness, influences the width of its BRDF's reflection lobe (which is visualized in Figure 2.8). In practice, for each pixel, based on its roughness value, one of the following approaches is used to calculate the reflection:

- *Roughness 0 - 0.3*: For specular (mirror-like) reflections, Lumen uses a ray tracing algorithm based on *stochastic screen-space reflections* by Tomas Stachowiak [Sta15]. This works by dividing the screen pixels into tiles. For each tile that needs to be processed, importance sampling of the visible reflection lobe is performed, and reflection rays are generated and sampled. To increase performance, the values obtained by these rays are reused for nearby tiles and weighted by their BRDF [WNK].

As expected, this introduces a lot of noise, which is mitigated by temporal filtering and further reduced by temporal anti-aliasing (TAA), if enabled. For areas with high variance and pixels revealed by disocclusion, a *bilateral filtering* is applied as well. For more details on bilateral filtering, please refer to Paris et al. [PKTD08].

- *Roughness 0.3 - 0.4*: Glossy reflections have a shorter ray distance, and if they miss all geometry, the closest-matching value from the nearest world space radiance cache is reused [WNK].
- *Roughness 0.4 - 1.0*: Reflections of roughest surfaces do not use additional ray tracing. Instead, the incoming radiance is interpolated from the screen space radiance cache based on the directions generated by importance sampling the GGX lobe [WNK].

  According to Wright [WNK], splitting the reflection pipeline into three parts reduces its overall cost by around 50-80%, based on the amount of reflective materials in the scene. Note that reflection rays work with both software and hardware ray tracing, but the hit-lighting pipeline of hardware ray tracing is recommended for the best results [Gamc]. Additionally, clear-coat reflections are supported with all ray tracing pipelines [Gam23, WNK].

  For more information on Lumen's reflections, please refer to Wright [WNK]. An illustrative image showing a scene with glossy materials can be seen in Figure 4.31.

**Figure 4.31:** An example of a scene rendered with both Lumen GI and reflections (source: [Gam23]).

## 4.10 Settings

Lumen offers a variety of settings, both project-wide and local only. These settings empower artists with the ability to adjust which parts of the rendering

process can utilize more computing power (e.g. samples, texel budgets, or time) or allow for disabling certain features entirely. All relevant settings can be found in the official Unreal Engine documentation [Gam23].

In Unreal Engine's editor, the project-wide settings can be found in *Edit -> Project Settings -> Engine -> Rendering*.

Unlike project settings, local settings are modified using a post-process volume object. During rendering, local settings are applied only when the camera is within the bounding box of this object.

Additionally, the post-process volume also allows to select local GI and reflection methods which will overwrite the project settings, thus allowing to entirely disable Lumen for certain parts of the scene.

Also note that apart from these public options, Lumen has a lot of hidden settings which can be accessed only through console commands. A presumably complete list of these commands is, as of January 2025, available on the following website: *https://forums.unrealengine.com/t/unreal-engine-5-all-console-variables-and-commands/608054*.

## 4.11 Lumen Analysis Summary

As demonstrated in this Chapter, Lumen is a very clever, complex, and customizable lighting pipeline. When set up correctly, it should, in theory, allow for real-time, high-quality global illumination for a large spectrum of different Unreal Engine scenes. I test whether this is true or not in the following Chapter.

Below is a short summary of the main concepts and techniques used by Lumen, together with references to their corresponding Sections [Gamc, Gam23, WNK, Wri21, Sko23, Unr21].

- First, Lumen stores a material parametrization of the nearby scene in various *surface cache* (4.7) texture atlases by orthographically projecting objects onto their pre-generated *surfel cards* (4.7.2). Furthermore, the surface cache stores radiance values, which are updated every frame to propagate object, camera, and lighting changes within this texture space (4.7.4). To accomplish this, Lumen uses shadow maps and shadow rays for direct lighting and radiance caching for indirect lighting. These update budgets are fixed for all frames but globally adjustable.

- Next, Lumen's *final gather* (4.8), based on *Monte Carlo integration* (2.4.5) utilizes a combination of *screen space tracing* (4.4) and a choice of either the *software* (4.5) or *hardware ray tracing* (4.6) pipelines (which have their own differently performant variants) to propagate radiance values from the surface cache to the pixels on screen, thus approximately solving the *rendering equation* (2.4.4). The software ray tracing pipeline traces against a *mesh distance field* representation (4.5.2) of individual objects, as well as a global, merged structure to accelerate intersection calculations at the cost of precision. In contrast, hardware ray tracing

keeps the original geometry, but is prone to performance issues in scenes where many objects overlap.

- To reduce noise and remain within the time constraints expected from real-time rendering algorithms, the radiance obtained by the final gather is heavily downsampled by using a combination of *screen space radiance cache* (4.8.1) for the first two meters and a *world space radiance cache* (4.8.2) for distant lighting. Ray generation uses *structured product importance sampling* (4.8.1) to super-sample the most important ray directions. After ray tracing is completed, the sampled incoming radiance values are *spatially filtered* (4.8.1) in the probe space, interpolated using *spherical harmonics* (4.8.3), and integrated at full resolution with BRDF and normal vectors stored in rasterized buffers.
- Finally, a *temporal filter* (4.8.4) is applied to remove artifacts, while screen space ambient occlusion is approximated using *bent normals* and composited to the final image in order to get back the contact shadows lost by operating in down-sampled resolution.
- If Lumen's *reflections* (4.9) are enabled, an pass is performed. Here, Lumen reuses data from the radiance cache to enhance rough and glossy reflections and traces additional rays for high-quality specular reflections.

# Chapter 5

# Testing Lumen

While I was mainly interested in discovering the algorithms and data structures that make Lumen work, a major portion of this thesis was dedicated to testing Lumen's visual quality, performance, and suitability for modern video games. In this Chapter, I will describe the process by which I was aiming to achieve this goal.

First, I created a showcase scene that could help me analyze demonstrate and how good the effects Lumen promises to achieve really look in practice, with three varying graphical settings and some additional toggle-able options. This part of the testing is covered in Section 5.2.

Second, I created a stress test scene, where the user can control the number of static and dynamic Nanite meshes and point lights in the scene and measure the average performance based on these quantities. A detailed description and results of this test are described in Section 5.3.

Afterwards, I created another scene in which the user can adjust the amount of overlapping, high-quality photo-scanned objects to further analyze the difference in speed between software and hardware ray tracing, as well as test Nanite's influence on Lumen's performance. For these results, see Section 5.3.4.

Finally, I tested Fortnite's performance, comparing the speed of Lumen's high and epic settings to rasterization, along with certain visual advantages or disadvantages both of these techniques offer in that particular game. My analysis of these results can be found in Section 5.4.

The controls of all test scenes are described in the Appendix B. The first two scenes were initially created in Unreal Engine 5.3, but I further improved them in Unreal Engine 5.5 and revisited all the results to come to the most accurate and relevant conclusions. All scenes were tested using the 1920x1080px resolution with no further upscaling.

Note that during this Chapter, despite the fact that 3D objects placed within an Unreal Engine scene are officially called *actors*, I will refer to them simply as "objects" for the purposes of clarity and consistency with the rest of this thesis.

## 5.1 Test Devices

Below you can find the hardware specifications of the three different devices I used for testing. Specifically, I used PC 1 and PC 3 to test my custom Unreal Engine scenes and PC 2 for testing Fortnite.

| | PC 1 | PC 2 | PC 3 |
|---|---|---|---|
| **Type** | Laptop | Desktop | Desktop |
| **OS** | Windows 10 Home 64bit | Windows 10 Home 64bit | Windows 11 Professional 64bit |
| **GPU** | NVIDIA GeForce RTX 3070 8 GB | NVIDIA GeForce RTX 2080 8 GB | NVIDIA GeForce RTX 4080 16 GB |
| **CPU** | AMD Ryzen 7 5800H 3.2 GHz | Intel Core™ i7-9700K 3.6 GHz | Intel Xeon W-2275 CPU 3.3 GHz |
| **RAM** | 2x16 GB RAM DDR4 3200 MHz | 2x8 GB DDR4 3200 MHz | 4x32 GB RAM DDR5 5600 MHz |

**Table 5.1:** Hardware specifications of all three devices I used for testing.

## 5.2 Feature Testing

My intention with the first test scene was to analyze how well Lumen handles the following features, which are advertised in Unreal Engine's documentation [Gam23] as some of its capabilities:

- **high-quality indoor indirect lighting**
- **hard and soft shadows**
- **emissive materials**
- **dynamic propagation of lighting and material changes**
- **multi-bounce mirror reflections**
- **light interaction with volumetric effects**

A skinned, rigged, and animated player character was also used in the scene to emphasize Lumen's limitations of illuminating dynamic meshes when using software ray tracing.

To analyze these effects, I set up a small opaque room with varying toggle-able objects inside and two sets of mirrors outside. An additional room, which contains the soft shadow testing, is placed next to it. The user can walk around the scene or cycle between a set of predetermined positions using the UI widgets. There, the user can also select one of the three following quality settings, which are managed by different Local Settings Volumes:

- **Lumen Default** — This is the default configuration that Unreal Engine provides. It uses the cache sizes and update budgets mentioned in the previous Chapter, and up to one mirror reflection bounce.
- **Lumen High** — For this configuration, I turned up every available setting to its maximum value. Specifically, the light-propagation update budgets are up to four times higher, and mirror reflections use eight bounces.
- **Lumen Off** — This setting disables Lumen completely. Unreal Engine's screen space reflections are used in its place to approximate mirror reflections, and only direct lighting is calculated.

Note that these configurations differ from Lumen's *High* and *Epic* settings used in Fortnite and their internal testing. Specifically, the Lumen Default still uses mesh distance fields first, before relying on the global distance field, unlike its Fortnite counterpart.

Both hardware ray tracing and Nanite can be optionally enabled using their corresponding checkboxes. These can be used in combination with any of the previously mentioned settings.

More details about the scene and its elements are covered in the following Sections, where I analyze each effect individually and highlight my most important findings. For clearer results, I disabled the user interface for all the screenshots I show here. All screenshots were captured in the Unreal Engine 5.5 version of the scene, unless stated otherwise. A complete gallery of image comparisons of different Lumen setups and PC configurations for all tests, created using the Unreal Engine 5.5 version of the scene, can be found in the Appendix C (inside the *Lumen Testing Results* document).

Finally, all meshes I used for this scene, including the player character, were taken from the Unreal Engine 5's 3rd person template scene.

### 5.2.1 High-quality Indoor Indirect Lighting Test

This test had three sub-objectives. First, I wanted to determine whether Lumen can light an indoor scene with only a very small amount of direct light, thus relying heavily on indirect lighting. My second objective was to evaluate the appearance of the diffuse color bleeding effect and how fast it updates in case the object's material color continuously changes. Finally, I wanted to see how much light leaking appears in a fully closed room with no direct lighting.

Lumen handled the first two problems reasonably well, as can be seen in Figures 5.1 and 5.2, but struggled significantly with the third one.

Specifically, the indirect lighting propagated immediately when the front wall of the room was moved. Except for the small amount of noise in the corners of the room, there were no noticeable artifacts in any of the setting configurations.

Similarly, when toggling on the dynamic material changes, the diffuse interreflection updates were near instantaneous, causing only a very brief

**Figure 5.1:** An example of an indoor scene lit primarily by indirect lighting. The skinned character properly receives indirect lighting due to screen traces, despite having no MDF representation and surface cache coverage. This image was rendered using Lumen Default and software ray tracing on PC 1.

mismatch between the object's color and its indirect effect on nearby environment. This delay was almost completely unnoticeable when using the Lumen High setting, as it updates a larger portion of the surface cache each frame.

The first major issue appeared when all direct lighting became blocked by the front wall, thus preventing any direct lighting from illuminating the room's interior. Instead of turning completely black, the room gradually faded to dark gray, while the corners remained unexpectedly bright, as shown in Figure 5.3. Further movement of the camera resulted in even stranger behavior, as the color of the corners started to change its tint to yellow. This was happening in the version of the scene built in Unreal Engine 5.3 and I assumed that it was a result of skylight leaking.

Surprisingly, the updated 5.5 scene has shown an even more curious behavior. The colors of the room corners no longer converged to yellow, but when using hardware ray tracing and the Lumen High quality setting, spots with blue tint often appeared at seemingly random places on the ceiling and the floor. These spots persisted across many frames until vanishing and reappearing again later, all without any camera or object movement.

Although the aforementioned slow convergence problem is probably caused by the surface cache updates, as they rely on indirect lighting from previous frames, I did not uncover the cause of the *blue spot artifact* problem, which can be seen in Figure 5.4. Since the skylight in the scene is more yellow than blue, I do not believe that leaking is the cause. My best guess is that the blue spots are caused by radiance cache's rays occasionally hitting the small emissive source located on the character's chest.

One supposed solution to these issues, mentioned in [Gamc] is to increase the values of the *Lumen Scene View Distance* and *Max Trace Distance* settings.

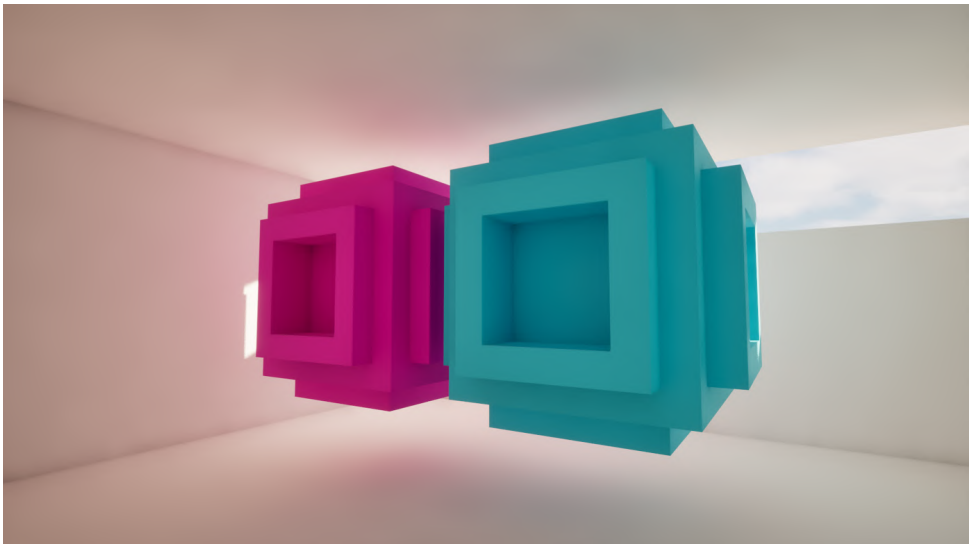


**Figure 5.2:** A subtle color bleeding effect caused by both objects can be seen on the ceiling, floor, and walls. This image was rendered using Lumen Default and hardware ray tracing on PC 1.

However, both of those attributes were set to their maximum allowed values for Lumen High, yet the problem persisted.

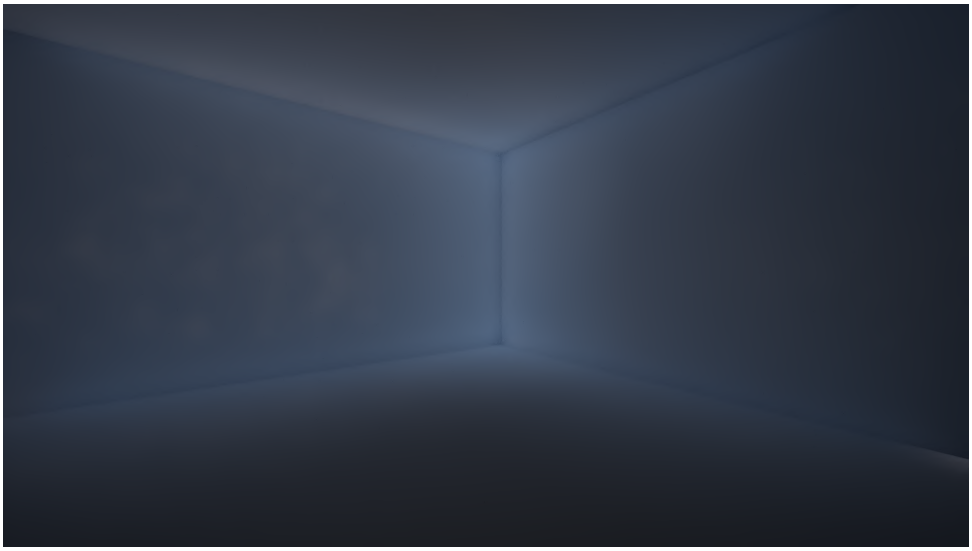


**Figure 5.3:** Apparent "Light leaking" seen in an unlit room. This image was rendered using Lumen Default and software ray tracing on PC 1 in Unreal Engine 5.3.

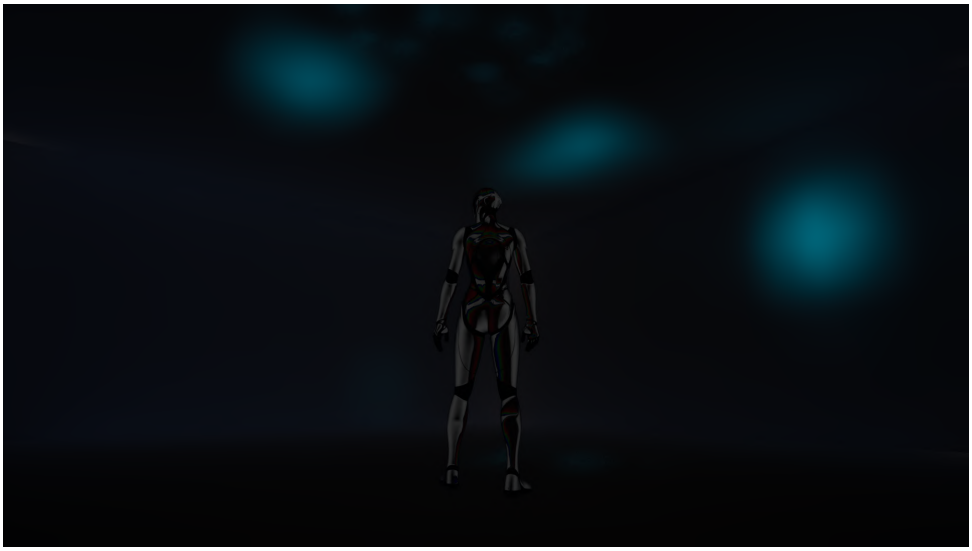


**Figure 5.4:** A strange blue spot artifact appearing in a closed, unlit room. This image was rendered using Lumen High and hardware ray tracing on PC 1.

### 5.2.2 Hard and Soft Shadows Test

To test Lumen's ability to produce various types of shadows, I set up a row of three diffuse cubes, where each of them is lit by a different light source positioned directly above it. The first of those light sources is a point light with no radius, the second is a point light with a radius of ten centimeters, and the third is a square-shaped area light. The rendered result of this part of the scene can be seen in Figure 5.5.

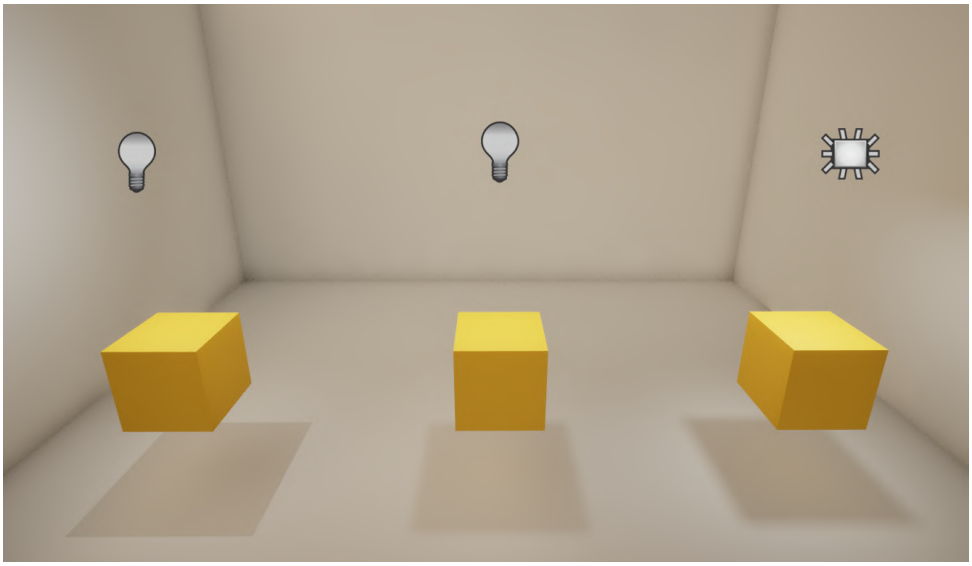


**Figure 5.5:** A showcase of three types of light sources correctly casting shadows with varying softness. This image was rendered using hardware ray tracing on PC 1 in Unreal Engine 5.3.

The difference in shadow types was noticeable, however increasing the width of the area light did not change the shadow's appearance (a standard behavior described in Section 2.5.1), which was a slight disappointment. Fortunately, changing the radius of the point light did influence the shadow border correctly across all tested configurations, with no noticeable differences in quality.

### 5.2.3 Emissive Materials Test

Lumen's official technical documentation states that emissive objects are supported, though they can supposedly produce a lot of noise and other artifacts if their size is too small. My intention was to test how efficiently a single average-sized emissive object can illuminate the whole room, with no other source of illumination present.

I was pretty impressed with the result, as the emission was not distractingly noisy and even without an additional light source overlaid on top of the emissive object (which is recommended in the official documentation [Gam23]), enough light was emitted so that all other objects were perfectly visible. This is shown in Figure 5.6.

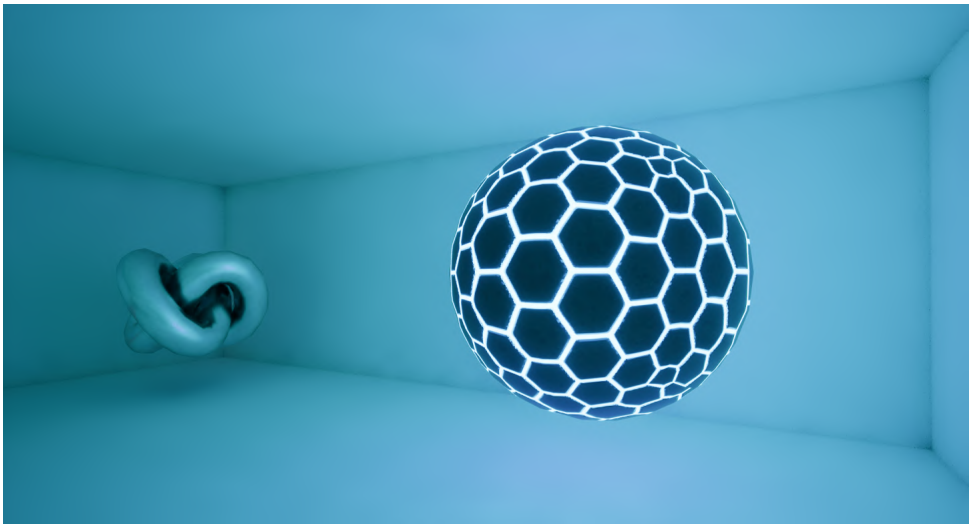


**Figure 5.6:** A room correctly lit by a visible emissive object. This image was rendered using Lumen Default and software ray tracing on PC 1.

Unfortunately, the lighting changes caused by disabling the emissive object propagated rather slowly, especially in cases where no direct skylight reached the room. For Lumen Default, it took around 15 seconds for the room walls to converge back to their expected (yet also inaccurate) dark gray colors. For Lumen High, the convergence time was noticeably faster, but not enough to not distract potential players. This issue (shown in Figure 5.7) was again likely caused by the limited surface cache update budgets, which are lower for indirect lighting. Since emissive materials are not considered light sources and are therefore not directly sampled, their effect on the scene is not immediate.

**Figure 5.7:** A room incorrectly lit by a disabled emissive object. This image was rendered using Lumen Default and hardware ray tracing on PC 1.

### 5.2.4 Light Propagation Test

Although dynamic diffuse bounce lighting was already tested in Section 5.2.1, I also wanted to see how objects and their shadows react to changes of the directional light. With this intention, I implemented a simple day and night cycle that can be toggled on and off at any time through the user interface. When enabled, the color and direction of the sunlight gradually change.

I was very pleased with the results, as the engine did not struggle at all with dynamically modifying the light source in real time. All of the shadows and directly lit areas were continuously being updated accordingly without any noticeable performance hits, pop-ups, or other artifacts, even on the Lumen Default setting.

### 5.2.5 Reflections Tests

Up to this point, I was generally pleased with Lumen's visual quality. The problems discovered with slow light propagation in certain edge cases (mentioned in Sections 5.2.1 and 5.2.3) were expected because of the way Lumen relies on the surface cache. Furthermore, I assume that most games will never even contain scenes with a complete lack of direct lighting and will thus evade these issues.

However, to evaluate the precision of Lumen's Reflections, I prepared four different subtests with results ranging from unimpressively unrealistic to dishearteningly disappointing.

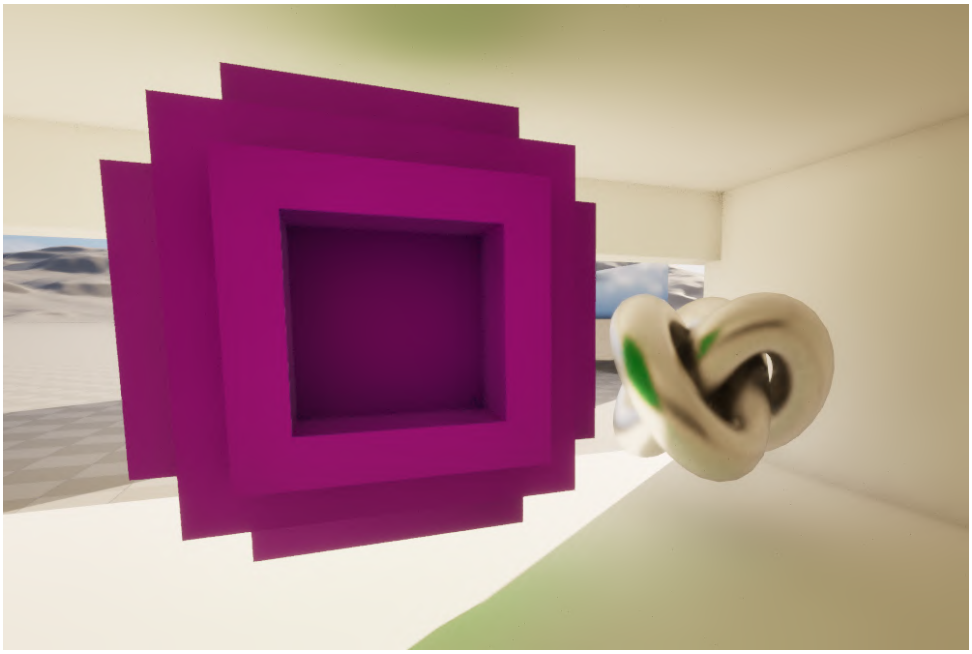


**Figure 5.8:** An apparent mismatch between a color of a diffuse object and its reflection in its neighboring reflective object. This image was rendered using hardware ray tracing on PC 1 in Unreal Engine 5.3.

### Reflection Update Speed Test

For the first test I used a static mesh with a glossy metallic material to assess how effectively it reflects light from nearby diffuse objects. Although it appeared to work correctly at first, when placed near an object whose color was continuously changing in time (as described in Subsection 5.2.1), the specular reflection seemed to update only at certain discrete intervals. This issue, once again caused by Lumen's fixed update budgets, often produced a distracting visual mismatch, which is shown in Figure 5.8. There, the diffuse object on the left has a pink tint, while its mirror reflection on the surface of the object to its right appears green, similarly to the color bleeding effect on the ceiling and the floor, which is less noticeable.

### Indoor Mirror Reflection Test

The second test involved covering the entire back wall of the test room with a mirror made of reflective material whose roughness was set to 0, which resulted in what was by far the most disappointing part of the entire feature testing. Not only was the reflection filled with a non-negligible amount of temporally incoherent noise, but the border between the part of the reflection which was able to use on-screen information and the part which had to rely on an additional ray tracing was clearly visible.

Furthermore, when using software ray tracing, the skinned character was almost completely missing in the reflection, and the clearly visible MDF

representation was very distracting. Selecting Lumen High had no perceptible visual impact despite having all reflection-related settings, such as quality, set to their maximum value.

Enabling hardware ray tracing improved the look of the diffuse objects, but some of their parts appeared as completely black. At first, I assumed this was due to an incomplete surfel card coverage of their meshes, but setting the number of cards to the maximum value did not eliminate the issue. This surprised me, as I would expect the default meshes provided by Unreal Engine to work correctly with their flagship feature. In addition, the noise was noticeably higher when using hardware ray tracing, which was a severe disappointment and the opposite of what I expected.

These results for software and hardware ray tracing are shown in Figures 5.9 and 5.10.

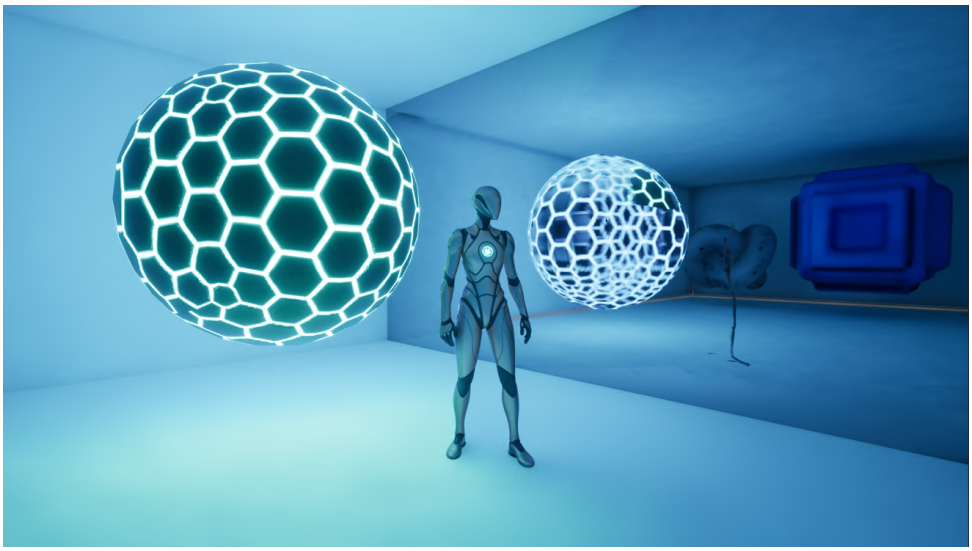


**Figure 5.9:** A showcase of indoor mirror reflections. Notice the low-frequency noise, almost entirely missing character, completely broken texture of the emissive object and the rough distance field representation of the blue object. This image was rendered using Lumen Default and software ray tracing on PC 1.

### Outdoor Mirror Reflection Test

In the previous test, I was disappointed by Lumen's inability to display correct reflections on a fully smooth mirror in an indoor setting. However, most of the materials used in video games are not fully specular and, as mentioned in Section 4.9, Lumen handles different roughness values with different methods.

To test the ability of Lumen's Reflections to correctly showcase glossy reflections, I placed a row of 10 additional blocks with roughness values ranging from 0.0 to 0.4 and a final, fully opaque block for comparison. All of them were placed outside the starting room. The expected difference between them was clearly visible. In addition, these mirrors seemed to look better than their inside counterpart, as the reflections were less noisy. Yet, this test

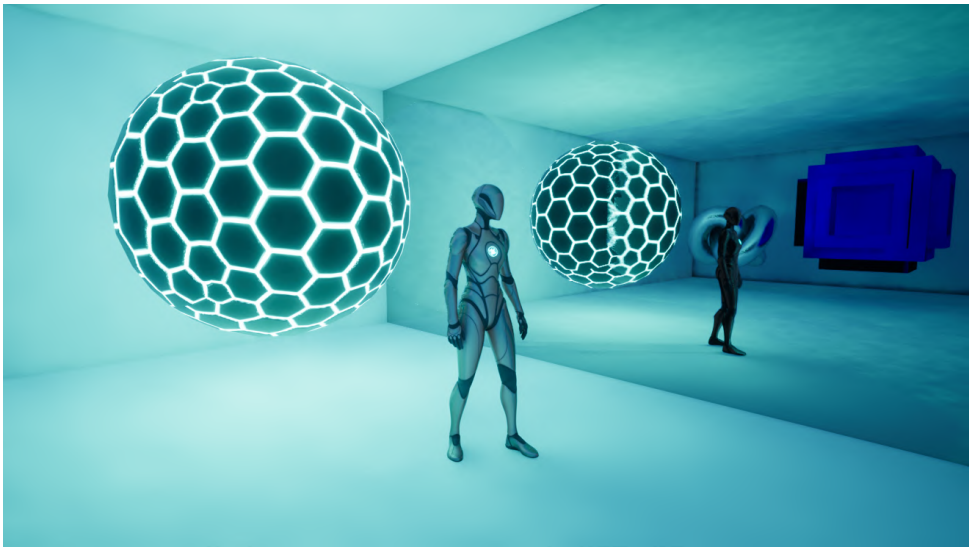


**Figure 5.10:** A showcase of indoor mirror reflections. Notice the almost entirely black character and other visual artifacts, such as the supposed partially missing surface cache coverage of the blue diffuse object. This image was rendered using Lumen High and hardware ray tracing on PC 1.

highlighted a different problem.

Parts of the default terrain mesh placed around the entire scene was apparently too far, as they were completely missing in the reflection when using the Lumen Default setting. Fortunately, it did appear correctly with increased ray tracing distance, which is the case when using Lumen High, as can be seen in Figure 5.11. Once again, this issue highlights a certain disconnect between assets provided by Unreal Engine as a starting point for new projects, and Lumen's initial settings.

Furthermore, despite the ability to solve this problem by switching to Lumen High for software ray tracing, enabling hardware ray tracing made the terrain reflection disappear once again, for reasons unbeknownst to me. My best guess is that Lumen does not incorporate terrain into its HLOD representation used for distant tracing or uses a separate distance limit for hardware ray tracing, which I do not find intuitive.

### Multi-bounce Mirror Reflection Test

Finally, I wanted to verify Lumen's support for multiple mirror reflection bounces. To test this, I placed two additional blocks facing each other.

As expected, the other mirror appeared to be completely diffuse in the reflection when using software ray tracing, as evident in Figure 5.12, except in cases where on-screen information could be used. On the other hand, hardware ray tracing combined with the Lumen High setting allowed for up to 8 reflection bounces.

However, I encountered more examples of undesired behavior. First, the resolution of skybox decreased with each reflection, which was very noticeable.


**Figure 5.11:** A comparison of outdoor glossy reflections with varying roughness rendered using Lumen Default and Lumen High with software ray tracing on PC 1. Notice the terrain missing in the reflection when using Lumen Default.

Furthermore, Lumen's reliance on screen space tracing for reflections caused many bizarre view-dependent artifacts, such as the one shown in Figure 5.13. More examples can be found in the feature testing image gallery. Fortunately, screen traces can be disabled, though it comes with a cost of completely losing the character in reflections.

Another unexpected issue I ran into occurred when positioning the camera directly next to a mirror and moving along its surface with motion blur enabled. This caused a very noticeable visual mismatch, where the reflection seemed to be drastically more blurred than the rest of the scene.

Note that switching to the hit-lighting pipeline (explained in Section 4.6.2) did not fix any of the aforementioned issues, despite it being presented as a way to enable higher-quality reflections in the documentation [Gamc].


**Figure 5.12:** A comparison of multi-bounce mirror reflections rendered using Lumen High on software and hardware ray tracing on PC 1. Notice the black character and downsampled skybox in reflections.

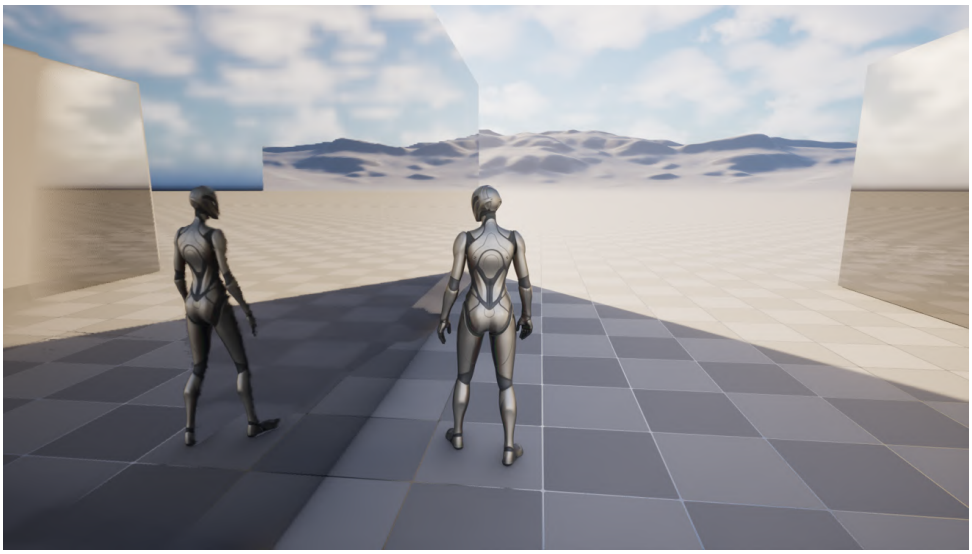


**Figure 5.13:** A showcase of many view-dependent artifacts caused by Lumen's Reflections. Notice the partially missing secondary bounce inside the mirror on left. The terrain is also missing in all reflections, except for the part where screen space tracing has usable data to work it. Part of the mirror reflection next to the character also appears incorrectly. This image was rendered using Lumen Default and software ray tracing on PC 1.

### 5.2.6 Volumetric Fog Test

I was interested in seeing how translucent volumetric effects interact with Lumen's global illumination. To test it, I used one of the new Unreal Engine 5.3 features - a *Local Height Fog* [Rea23]. It is an object that can be placed within a scene and has a configurable size, density, color, scattering distribution, and shape. I placed it within the enclosed starting room.

The result was not very impressive as there was a great amount of light leaking despite the fact that the sunlight should have been completely occluded by walls and the ceiling. This artifact, shown in Figure 5.14, was much more noticeable when using higher values of the scattering distribution.

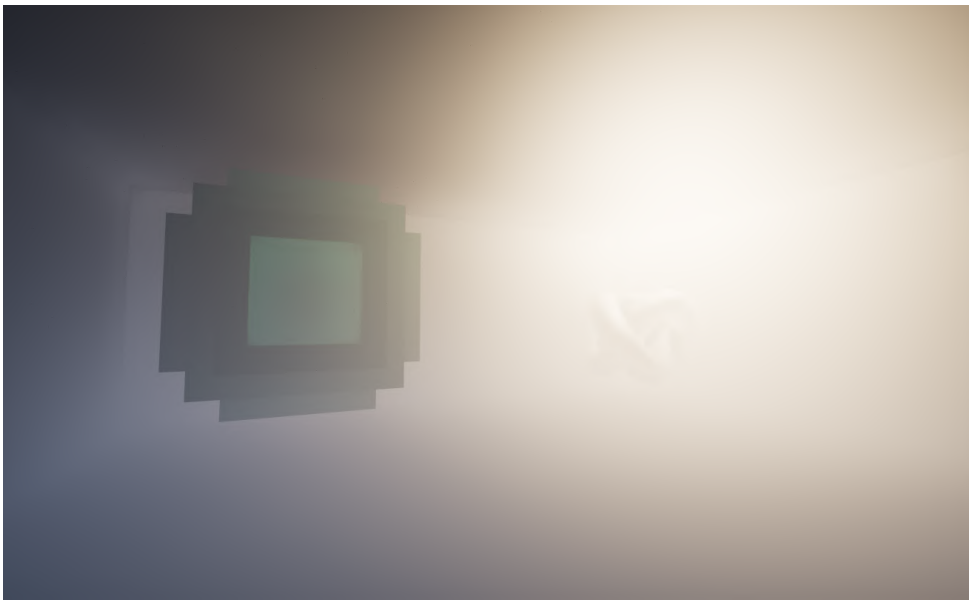


**Figure 5.14:** A massive light leaking issue caused by a local volumetric height fog with scattering distribution of 0.8. This image was rendered using software ray tracing in Unreal Engine 5.3.

However, this was not surprising, as Daniel Wright has previously stated that rendering volumetrics and their interaction with GI is an area they wish to improve in future Unreal Engine releases [Wri21].

Unfortunately, I was unable to replicate this test in Unreal Engine 5.5, since the local fog became invisible when the camera was near it. For this reason, I removed it from the final test scene.

### 5.2.7 Feature Testing Summary

To summarize, despite Lumen's ability to approximate global illumination rather well, it seemingly struggled to avoid visual artifacts in certain edge-case situations, mainly in completely occluded indoor areas. I was pleasantly surprised by how seamlessly Lumen handled dynamic lighting changes, despite some unexpected behavior when toggling emissive objects. Furthermore, since

this test scene contained only very simple meshes, I was unable to notice any visual advantage obtained by using hardware ray tracing or Nanite.

The biggest disappointment for me were definitely Lumen's reflections, as I was unable to make them look even remotely realistic and noise-free, even with hardware ray tracing and the hit-lighting pipeline (which, when enabled, seem to have virtually no impact on the visual quality, despite the official sources I used claiming otherwise). Although this issue may have been caused by me not finding the optimal configuration of the available settings, I believe that default values should always provide a good looking ratio of visual appearance to performance. This certainly did not apply here. Similarly, when most settings are set to their maximum values, as was the case with Lumen High, I expect a reasonable improvement in visual quality, yet most issues persisted. Disabling screen tracing removed some of the view-dependent visual inconsistencies at the cost of losing the dynamic character mesh in the reflection, making it a choice between what type of artifact was I willing to accept. Ultimately, I came to the conclusion that noisier image is less distracting than view-dependent lighting, thus I disabled screen tracing in the Lumen High setting for the performance testing scene described in the next Section.

As previously mentioned, Unreal Engine allows for a lot of customization and configuration through console commands. Currently, 2471 commands relate to rendering settings. More than 200 of those directly alter Lumen's behavior, while many of the remaining ones influence the data structures it uses, such as distance fields. I do believe that it is possible to eliminate the majority of issues I encountered during this testing via a careful combination of these console commands, but the scope of their documentation is very limited. Furthermore, using these commands requires a detailed understanding of Lumen's individual components, such as the surface cache or final gather, which is in my opinion unreasonable to expect from all artists. This is further underlined by the fact that official sources such as Unreal Engine's documentation page about Lumen's technical details ([Gamc]) or Daniel Wright's video about Lumen's high-level overview ([Unr21]) do not even mention the final gather at all.

Note that these sources do provide tips that can help alleviate some common visual artifacts, but unfortunately they did not solve the specific issues I encountered. More evidence for this claim is shown in the next Section 5.3.5. For these reasons, I believe that as of now, many developers may have a difficult time adapting Lumen to their specific use cases without encountering any issues.

Finally, testing the scene on a high-end device (PC 3) had no perceptible impact on visual quality, except for the increased framerates, which was expected.

## 5.3 Custom Performance Testing

After verifying that Lumen does indeed enable real-time dynamic global illumination and reflections, though not with a perfect quality, I was eager to test its performance with regard to the complexity of the rendered scene. This includes the influence of object counts, point light counts, and dynamic object counts.

### 5.3.1 Performance Testing Scene

For this purpose, I created another Unreal Engine scene, called *Performance Testing*, which contains a large square-shaped floor surrounded by four walls. The user can select the number of Nanite meshes and point lights contained within this bounded area. Each of these meshes has around a 1000 triangles and is randomly translated, scaled, and rotated in such a way that minimizes potential overlap. A subset of these meshes, limited to 20 000, can be parented to a continuously rotating object, making it dynamically traverse the scene. Furthermore, the user can decide to alter the angle of the directional light, toggle it entirely and change the material of the walls from opaque to fully reflective. Similarly to the feature testing scene, the user can also choose between the Lumen Default and Lumen High setting configurations, as well as whether to use software or hardware ray tracing.

Since Lumen's pipeline is rather complex and split between different functions which operate on both CPU and GPU, I mostly focused on measuring the overall time it takes to render each frame, along with the average frames per second, as that is a statistic most video game players will ultimately be interested in. Once the user configures the scene, they can obtain these measurements by pressing the $t$ key, which starts the testing and disables interaction for 20 seconds. During this period, the camera follows a predetermined path to minimize the impact of view-dependent changes in geometric complexity and measures the average frame time.

The results of my tests using this scene are shown in Section 5.3.5 and an illustrative screenshot can be seen in Figure 5.15. All Nanite meshes I used are publicly available for free in the Quixel mega-scan library at FAB (*https://www.fab.com/sellers/Quixel*).

### 5.3.2 Overlap and Nanite Testing Scene

Unreal Engine 5.5 has many possible settings that can drastically influence frame times, such as the previously mentioned Nanite or the new experimental feature called *Mega Lights*, which, according to the official documentation, severely improves the speed of calculating direct lighting for scenes with many point lights [Epi24].

Testing all possible configurations of these settings within the Performance Testing scene did not seem feasible to me. Therefore, I decided to first measure Nanite's impact on Lumen using a third, simpler scene. Here, the user can


**Figure 5.15:** A showcase of the Performance Testing scene containing 100 000 objects, 255 point lights and no directional light.

spawn photoscanned meshes consisting of ten thousand triangles. These meshes are spawned in such a way that there is a high chance they will overlap, which allowed me to verify Wright's claims that hardware ray tracing struggles with overlapping objects, unlike its software counterpart. This scene is called *Overlap Testing* and its performance is measured similarly to the aforementioned Performance Testing scene, with two small differences: the camera is stationary and the testing duration is limited to 10 seconds. The results of this test are discussed in Section 5.3.4, and an illustrative screenshot of the scene can be seen in Figure 5.16.


**Figure 5.16:** A showcase of the Overlap Testing scene containing 10 000 objects.

### 5.3.3 Used Abbreviations

In the following Sections, I present some of the most interesting results I measured on PC 1 and PC 3 in both of the test scenes along with my conclusions. For a complete list of results, please refer to the *Lumen Testing Results* document found in the Appendix C.

Note that for compactness, the following abbreviations of various settings are used in all Figure descriptions:

- *SWRT* - Software Ray Tracing
- *HWRT* - Hardware Ray Tracing
- *LD* - Lumen Default
- *LH* - Lumen High
- *N* - Nanite
- *ML* - Mega Lights

Furthermore, keep in mind that the object thresholds I chose for all tests are not evenly spaced. Instead, I selected them in order to simulate scenes with varying but reasonable levels of complexity. For the Overlap Testing scene, these values were ranging from representing an almost empty scene with 1 000 objects, through medium-sized scenes (10 000, 50 0000, and 100 000 objects) to heavily detailed with 200 000 objects. For Performance Testing, I raised the final tested value to 250 000. In order to preserve the information clarity, I linearized all the graphs. Note that only the highlighted values at the named thresholds were actually measured.

### 5.3.4 Overlap and Nanite Testing Results

As mentioned multiple times in Chapter 4, Nanite plays a crucial role in accelerating surface cache recaptures, making it up to ten times faster. For this reason, Epic Games recommends to always use Nanite meshes together with Lumen. However, I wanted to measure exactly how Nanite affects Lumen with different object counts [Unr21].

I selected eight configurations of settings to test, four of which had Nanite enabled. For each of the configurations, I used the Overlap Testing scene to spawn increasing amounts of high-poly photoscanned meshes. The results obtained on PC 1 are shown in Figure 5.17, where we can see that above the 50 000 object threshold, the average frame time of all configurations without Nanite increased very fast beyond what is suitable for real-time applications, reaching hundreds of milliseconds.

Furthermore, with the higher object counts, software ray tracing with Nanite enabled was around twice as fast as its hardware counterpart. One exception was the Lumen High setting, where the frame time of hardware ray

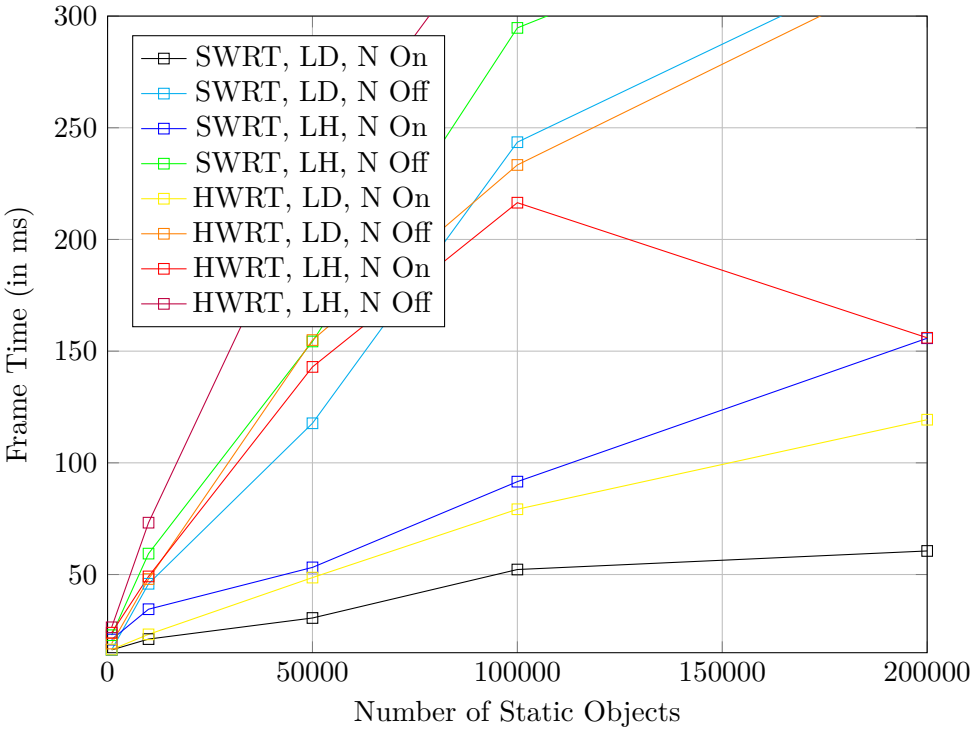



**Figure 5.17:** Overlap and Nanite test results with 8 various settings, measured on PC 1.

tracing (highlighted in red) decreased significantly above the 100 000 object threshold, nearly matching the corresponding software configuration.

Moving on to the results obtained on the more powerful PC 3 (shown in Figure 5.18), we can see similar patterns and relative differences between all configurations, although the frame times are substantially lower across the board. Surprisingly, when using software ray tracing, Lumen Default was around 20 ms slower than Lumen High with 100 000 objects, possibly due to its lack of screen space ray tracing.

Based on these results, I decided to use Nanite for all further testing, as its positive influence on performance was undeniable, decreasing the total frame times by up to 150 ms. Furthermore, I verified that hardware ray tracing gets slower as the number of overlapping objects in the scene increases, more so than software ray tracing.

This is mostly visible when comparing configurations which used Lumen Default with Nanite enabled. With 200 000 objects in the scene, software ray tracing had an average frame time of 60.55 ms on PC 1 and 32.13 ms on PC 3. Hardware ray tracing reached 119.31 ms on PC 1 and 170.56 ms on PC 3, which is two to six times slower.

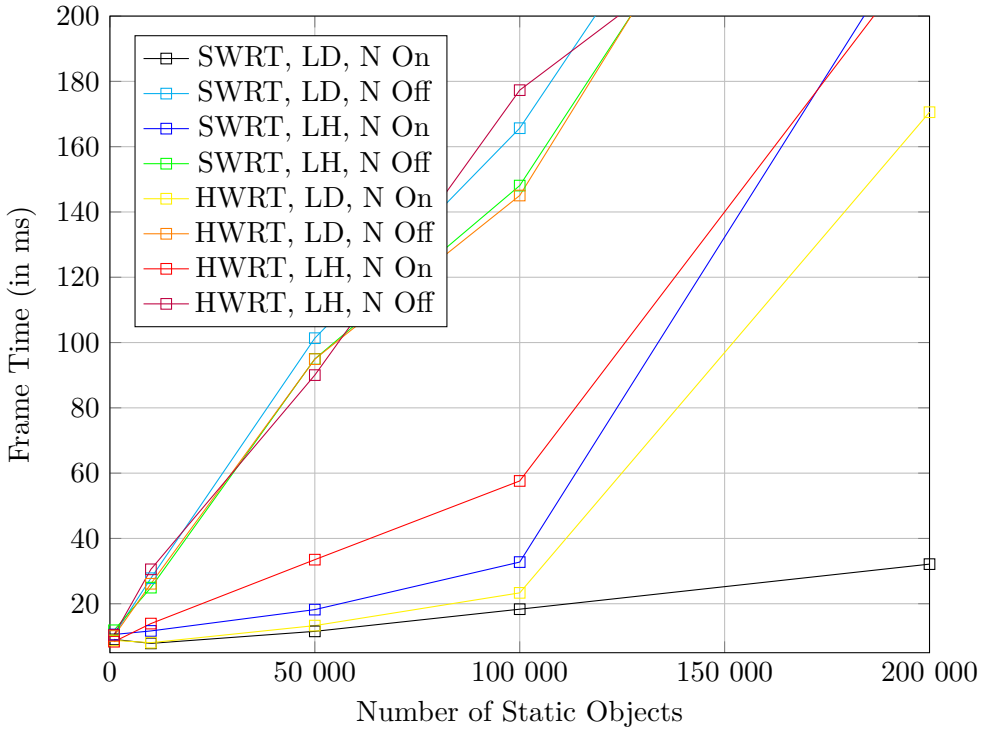



**Figure 5.18:** Overlap and Nanite test results with 8 various settings, measured on PC 3.

### 5.3.5 Performance Testing Results

While the results obtained from the Overlap Testing scene hinted at a speed difference of hardware and software ray tracing in a certain scenario, scenes with lots of overlapping objects are not recommended for Lumen. Therefore, one of my main intentions with the Performance Testing scene was to compare the two pipelines when most of the objects do not overlap, as it is supposedly the more common situation. However, this was not my only objective with the Performance Testing scene. The others are listed below and further analyzed in their corresponding Subsections:

- Measuring the impact of Mega Lights on performance when the scene contains many point lights.
- Measuring the performance trade-off between the Lumen Default and Lumen High configurations.
- Measuring the effect of enabling fully reflective walls, thus forcing Lumen to calculate mirror reflections.
- Measuring the impact of increasing amounts of dynamically moving objects.
- Measuring the performance increase obtained from the use of more powerful hardware.

For these reasons, I decided to test only the following configurations on PC 1, as I found them to be the most crucial and illustrative.

1. software ray tracing:
   a. Lumen Default + opaque walls
   b. Lumen Default + reflective walls
   c. Lumen High + opaque walls
2. hardware ray tracing:
   a. Lumen Default:
      (i) Mega Lights disabled + opaque walls
      (ii) Mega Lights enabled + opaque walls
   b. Lumen High:
      (i) Mega Lights disabled + opaque walls
      (ii) Mega Lights enabled + opaque walls
      (iii) Mega Lights enabled + reflective walls
3. dynamic object influence:
   a. SWRT, Lumen Default, Nanite on, Mega Lights off, 0 point lights
   b. HWRT, Lumen High, Nanite on, Mega Lights on, 100 point lights

For the second testing (performed on PC 3) I omitted the configurations which used hardware ray tracing without Mega Lights as after the PC 1 testing, I came to the conclusion that this feature is necessary in order to achieve noticeably better performance, similarly to Nanite. However, I disabled Mega Lights for all tests with 0 point lights, as it would likely introduce an unnecessary overhead with no benefits [Epi24].

### Mega Lights

As previously mentioned, Mega Lights is a new experimental feature introduced in Unreal Engine 5.5, which uses ray tracing and occlusion importance sampling with a fixed budget to solve all direct lighting. Its official documentation [Epi24] contains a very detailed description of how it works and how it is supposed to be used in order to minimize errors and visual artifacts. Specifically, it recommends using hardware ray tracing for optimal performance. Therefore, I did not enable Mega Lights for any of the configurations which used software ray tracing.

The results obtained from my first testing (visualized in Figure 5.19) show that more than doubling the number of point lights in the scene had a relatively small impact on the performance when using Mega Lights. For example, with 50 000 objects in the scene, the frame time increased only by 2.17 ms when adding 155 more point lights. However, without Mega Lights,

the frame time, which was already twice as high, increased by 43.82 ms, nearly halving the average FPS.

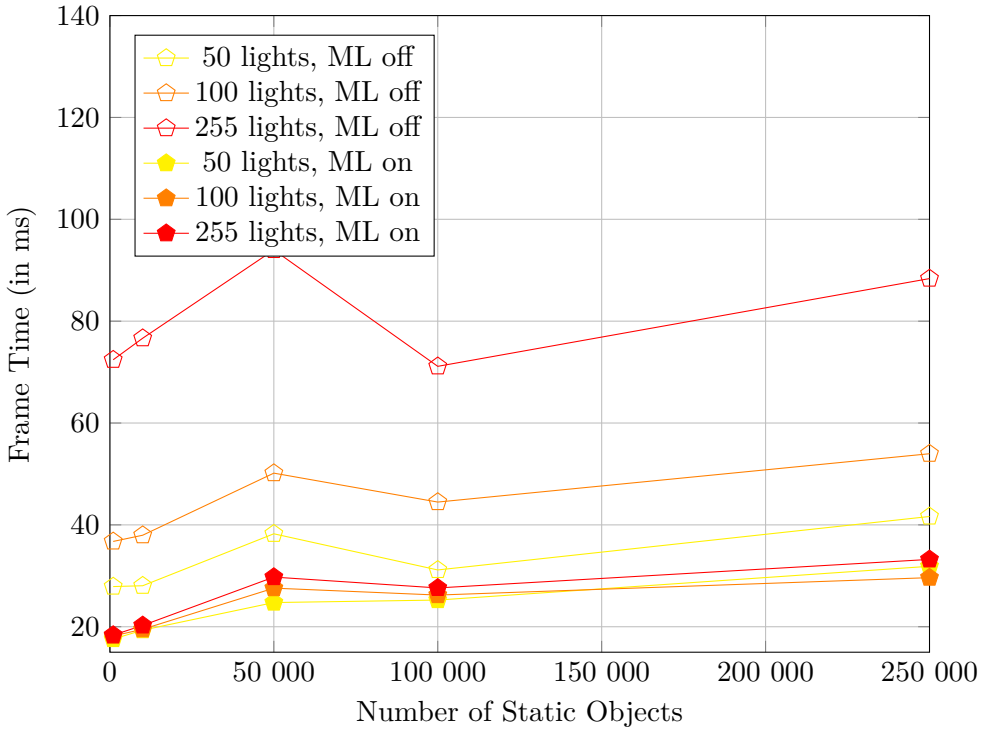



**Figure 5.19:** A comparison of results for 50, 100, and 255 lights with ML on and off, measured on PC 1 with HWRT, Lumen Default, and opaque walls.

Note that the noticeable frame-time drops caused by jumping from 50 000 objects to 100 000 objects were probably caused by culling, as the amount of occluded screen space at any given point increased significantly. This drop was severely less noticeable when switching to Lumen High (shown in Figure 5.20). Here, we can see that configurations with Mega Lights are still noticeably faster, though all average frame rates are significantly higher. It is worth noting that with 250 000 objects and 255 point lights, Mega Lights did have the most noticeable increase in frame time, though the result was nonetheless lower by around 60 ms than when disabled.

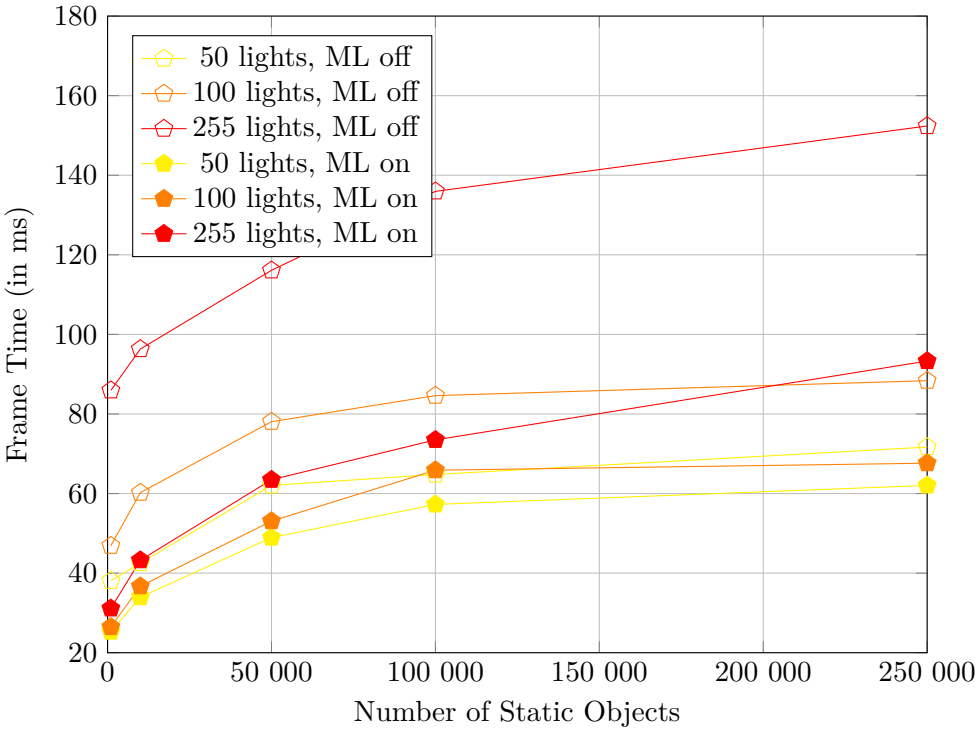



**Figure 5.20:** A comparison of results for 50, 100, and 255 lights with ML on and off, measured on PC 1 with HWRT, Lumen High and opaque walls.

Based on these results, I believe that once Mega Lights is finished and no longer just an experimental feature, it should be used in all scenes with high number of shadow-casting points lights, as the performance increase it provides is nothing less than substantial. Furthermore, after enabling Mega Lights, I did not experience any perceptible loss in visual quality. Therefore, I do not recommend having hundreds of point lights in scenes with thousands of objects without using Mega Lights unless real-time framerates are not desired.

### Software vs Hardware Ray Tracing

After reading most of the available official sources related to Lumen, I expected hardware ray tracing to be substantially slower than software ray tracing in almost all cases. However, in Unreal Engine 5.5, Epic Games optimized hardware ray tracing making it target 60 FPS across all supported hardware configurations, which was previously the case only for software ray tracing [Gam24c].

Although I was unable to find out what specific changes the engineers at Epic Games implemented, the results I measured on both PC 1 (shown in Figure 5.21) and PC 3 (shown in Figure 5.22) confirm that their intentions were likely successful.

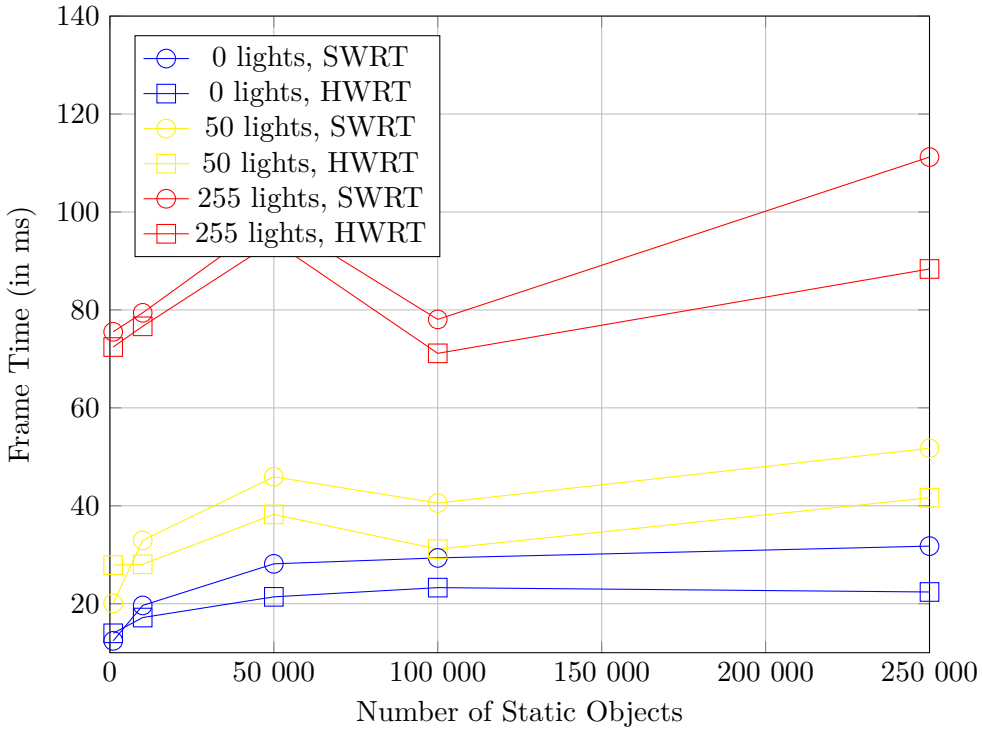



**Figure 5.21:** A comparison of software and hardware ray tracing results for 0, 50, and 255 lights with ML off, measured on PC 1 with Lumen Default and opaque walls.

To be more specific, the first graph shows that on PC 1 and without Mega Lights enabled, hardware ray tracing was consistently faster by around 2 to 5 ms, which is not a negligible amount. The configurations with 1 000 objects and low numbers of point lights are an exception and should be considered. Unfortunately, once the number of lights reached 255, the frame times were so high that I could no longer consider them suitable for real-time applications, especially video games. This gave me the impression that despite the improved hardware ray tracing, Lumen still needs more time to be optimized for larger and more complex scenes, as even with no point lights, the 60 FPS target was not achieved.

However, once I tested the same configurations on PC 3, which has a significantly more powerful GPU, I was pleasantly surprised by the results. Furthermore, enabling Mega Lights did wonders for performance, as can be seen in Figure 5.22. Not only was hardware ray tracing with 255 lights mostly faster than software ray tracing with 0 lights, but all configurations managed to render at below 12 ms per frame. Specifically, when averaging all results for all point and object count configurations, software ray tracing run at around 105.25 FPS, while hardware ray tracing was able to reach 117.70 FPS. This is almost a double of what Epic Games set out to achieve, and as new generations of graphic acceleration units continue to get developed, I believe that Lumen's hardware ray tracing will only get faster.

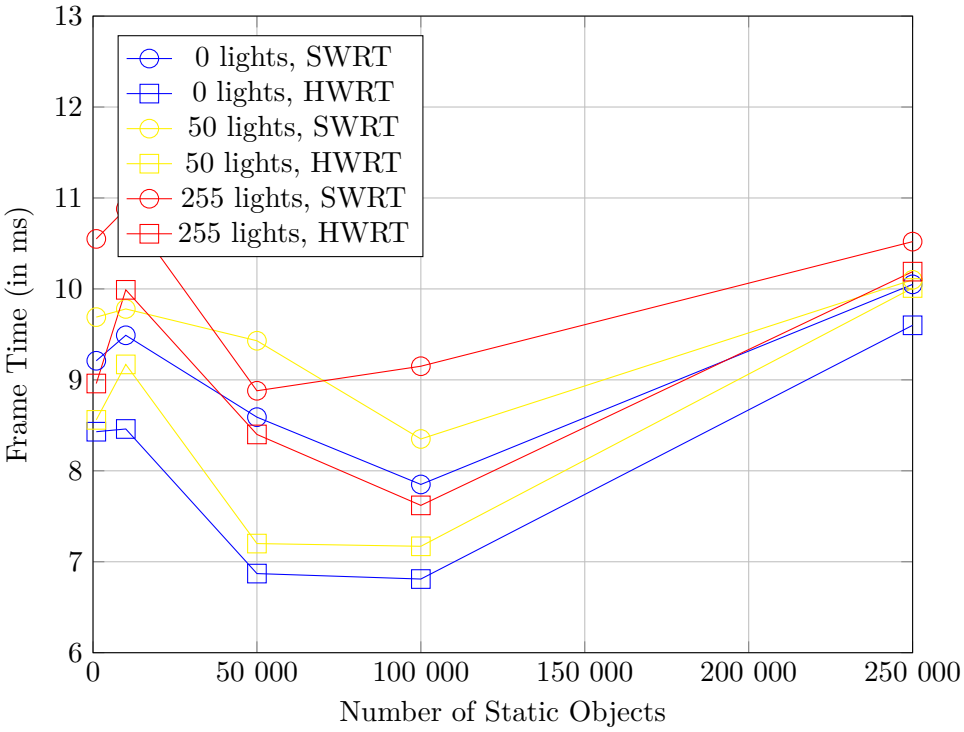



**Figure 5.22:** A comparison of software and hardware ray tracing results for 0, 50, and 255 lights with ML on, measured on PC 3 with Lumen Default and opaque walls.

### Lumen Default vs Lumen High

I created the Lumen High configuration in order to push Lumen's limits and see whether its almost unlimited ray distance, increased surface cache resolution, and final gather update budgets severely impact the performance or not. Since artists will hopefully not use a configuration such as this for older hardware and without further optimizations, I will highlight only the results I measured on PC 3 with hardware ray tracing and Mega Lights enabled (illustrated in Figure 5.23).

Here, we can see that as the number of point lights in the scene increased, Lumen High suffered noticeably larger jumps in frame times than Lumen Default. This is most evident when the scene contained 50 000 and 100 000 objects respectively, since the performance of Lumen Default actually increased for reasons I already mentioned in Section 5.3.5. This was not the case for Lumen High. Despite this, Lumen High still managed to render the scene at more than 60 FPS (and achieved a total average of 81.16 FPS), with the exception of the configuration using 255 lights. I find that rather impressive, as in high-quality visualization scenes, improved detail and fast light propagation are important and might be worth this additional performance cost.

Suffice to say, the setting configuration provided by Lumen Default will likely be enough for most projects, as it strikes a good balance between

performance and the visual quality of its global illumination.

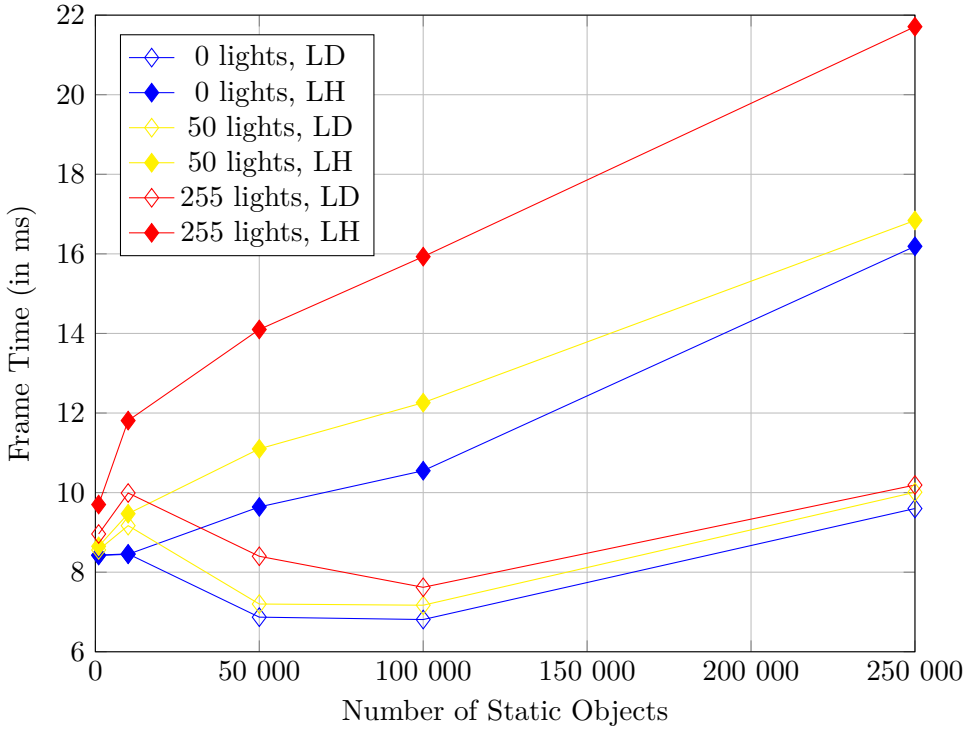



**Figure 5.23:** A comparison of Lumen Default and Lumen High results for 0, 50, and 255 lights with ML on, measured on PC 3 with hardware ray tracing and opaque walls.

### Reflections Performance

While I was unimpressed by the limitations of Lumen's mirror reflections (as thoroughly explored in Section 5.2), I was nonetheless interested in measuring the impact of this additional part of Lumen's pipeline on its rendering speed. Since Lumen Default and High have different maximum of computed reflection bounces, I will highlight results of 4 settings configurations in total, all measured on PC 3.

Figure 5.24 shows the data obtained from the 2 configurations using software ray tracing and Lumen Default, which is limited to 1 mirror reflection bounce. We can see that enabling the reflective walls had a larger impact as the number of lights in the scene increased, particularly when the scene contained 50 000 objects and 255 lights. However, the total difference was only around 1 ms, which is less than I expected. In all other scenarios, reflective walls had even a lesser impact on performance, to the point of being almost negligible.

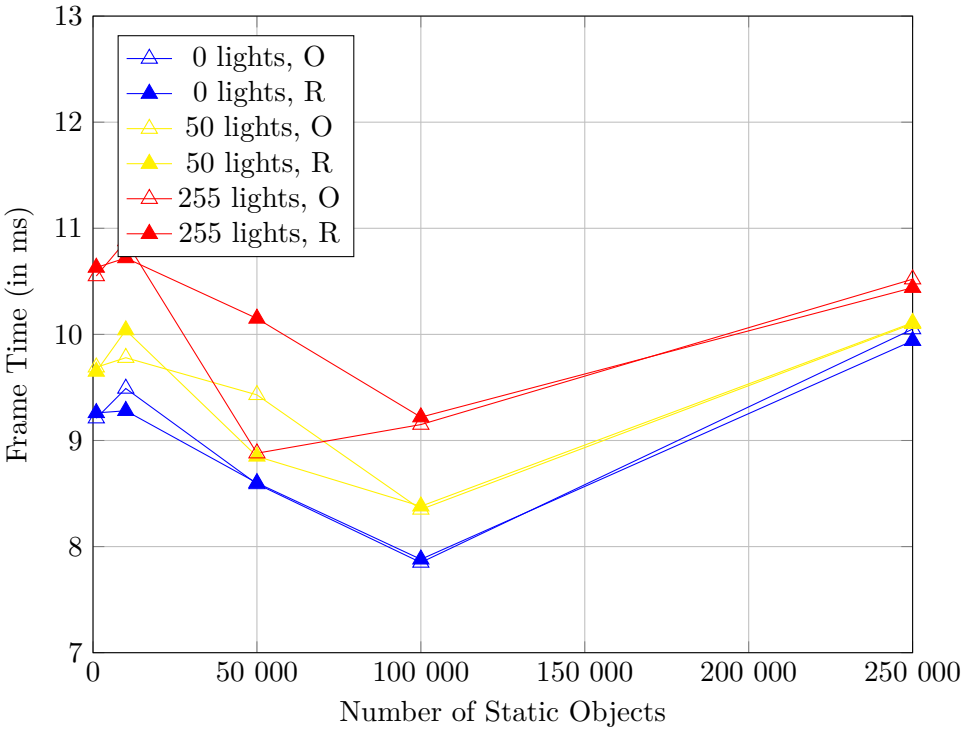



**Figure 5.24:** A comparison of results with opaque (O) and reflective (R) walls, using 0, 50, and 255 lights with ML off, measured on PC 3 with Lumen Default and software ray tracing.

Interestingly, once the scene contained 250 000 objects, the configuration that used reflective walls actually performed faster by 0.06 ms. I would assume this to be just a statistical error caused by a low number of measured samples, but further testing indicates that this might not be the case.

Specifically, the results measured using hardware ray tracing with Lumen High and Mega Lights on show an even higher increase in frame time for the configuration that used opaque walls when there were 250 000 objects in the scene, regardless of the light count. The exact average difference was 3.17 ms. This is shown in Figure 5.25, where we can also see that with lower object counts, opaque walls were always faster to render, as expected.

Truthfully, I do not understand why this phenomenon occurs, since, as we explored in Chapter 4, Lumen uses additional ray tracing steps for pixels belonging to materials with roughness less than 0.4. With no such pixels, as is likely the case with 250 000 objects scattered around the scene, I expected both settings to have the same render time at best, yet that was not the case.

When omitting this outlier and averaging the rest of object counts, reflective walls with 8 light bounces cost around 1.66 ms to render, which is still reasonably fast and fits within the budget of 60 FPS.

Note that having a scene which contains mirrors as large and smooth as the walls within the Performance Testing scene is definitely not common, and as such, I would assume that Lumen's reflections are very well optimized for typical use cases, such as small indoor mirrors or puddles of water.

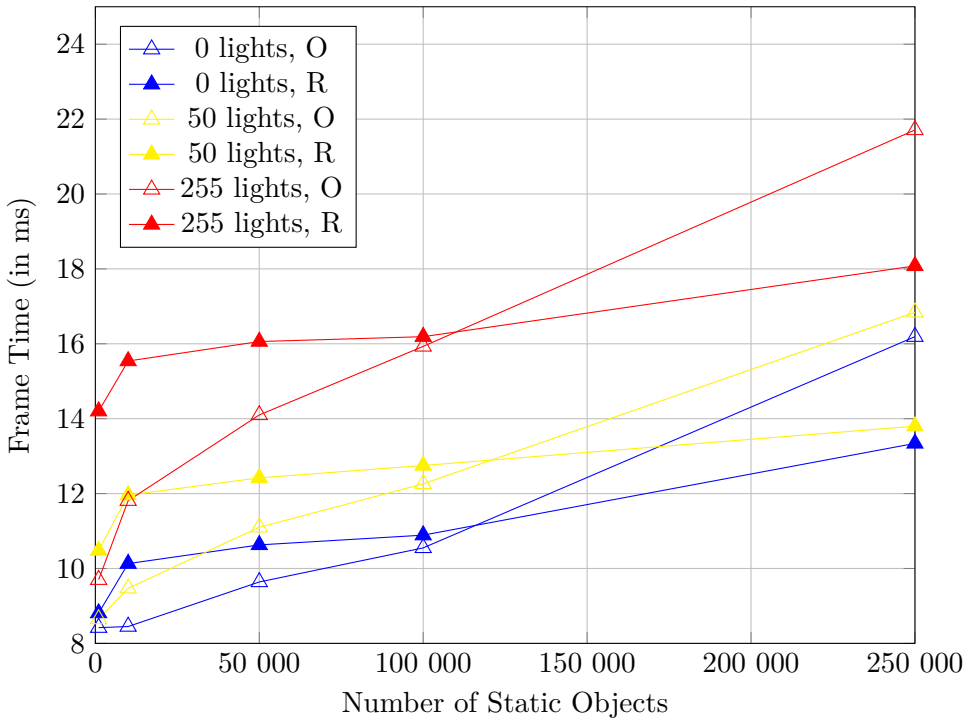



**Figure 5.25:** A comparison of results with opaque (O) and reflective (R) walls, using 0, 50, and 255 lights with ML on, measured on PC 3 with Lumen High and hardware ray tracing.

### Dynamic Objects Performance

Finally, I was interested in the impact of dynamic objects. Including them in the scene forces software ray tracing to update the global distance field, while hardware ray tracing needs to rebuild its BVH. I used the following two configurations of settings:

- Setting 1 – SWRT, Lumen Default, Mega Lights off, 0 point lights
- Setting 2 – HWRT, Lumen High, Mega Lights on, 100 point lights

I combined the results obtained on PC 1 and PC 3 into a single graph shown in Figure 5.26. Here, we can see that increasing the dynamic object count had a similarly large impact on both settings, though my guess was that hardware ray tracing would be affected severely more.

What I find more interesting is that PC 3 struggled far more once the number of dynamic objects reached 10 000. To be more precise, the average frame time of PC 3 was around 30 ms higher than PC 1.

Furthermore, PC 1 rendered the scene at nearly the same frame rates for both 1 000 and 5 000 objects, while the frame time of PC 3 nearly doubled. This was likely not caused solely by Lumen however, as changing the object's transformation is presumably a task that runs on a single CPU thread, which has a faster base clock speed on PC 1, thus possibly justifying the difference in performance.

Nevertheless, I would not recommend using more than a few hundreds of dynamic objects with Lumen, as the underlying acceleration structures work best for largely static scenes.

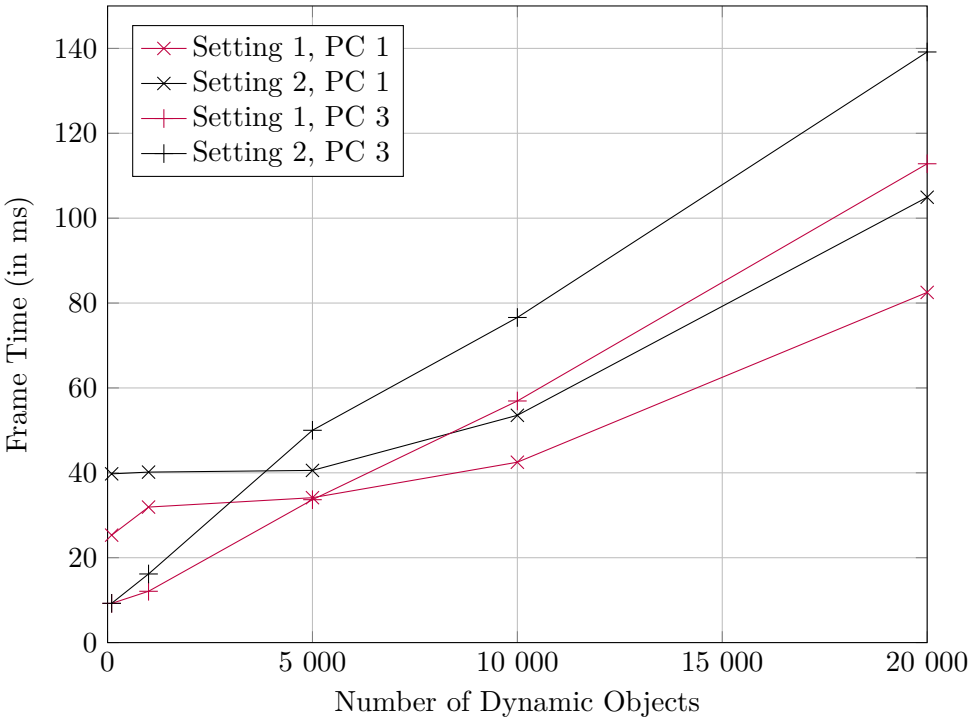



**Figure 5.26:** A comparison of results obtained from testing the influence of dynamic objects using Setting 1 and Setting 2 on PC 1 and PC 3.

## Visual Artifacts

During the Performance Testing, I encountered a few more visual artifacts that I found interesting.

First, when using Lumen High and hardware ray tracing, most objects were not visible in the mirror reflections on walls. This issue can be seen in Figure 5.27, and is likely caused by the limited memory of the surface cache. Realistically, this is also an issue caused by having an absurdly large mirror which reflects most of the scene. And while the official documentation [Gamc] indicates that this issue can be mitigated by increasing the surface cache's memory, Lumen High already uses the maximum allowed value, yet the problem persisted.

Second, when using hardware ray tracing, the image appeared to lack indirect shadowing. This is shown in Figure 5.28 and I am unsure as to why this is caused, or whether it is even an undesirable behavior. Subjectively speaking, I found software ray tracing to produce a better looking global illumination in this scene, which was surprising.

**Figure 5.27:** A screenshot of the Performance Testing scene with 18 000 objects. Notice how the majority of objects does not appear in the mirror reflection on left. Furthermore, some of those which do appear have missing surface cache coverage, thus partially showing as black. This image was rendered using hardware ray tracing with Lumen High, Nanite and the Hit-Lighting pipeline.


**Figure 5.28:** A visual comparison between hardware and software ray tracing. Rendered with Nanite and Lumen Default.

### 5.3.6 Performance Testing Summary

Overall, the results I measured tend to mostly line up with the goals and descriptions that Epic Games provides us in the official documentation [Gam23], even surpassing them on new hardware (as was the case with PC 3).

That is, Lumen can clearly hit the 60 FPS mark, albeit for more complex scenes, it is consistent only when relying on Nanite and Mega Lights. This is

not as big of an issue, as most of the mesh types work correctly with Nanite by default, and enabling Mega Lights is barely an inconvenience. Based solely on these results, I believe that with new, more powerful hardware, Lumen will have little to no trouble maintaining its real-time performance for years to come.

However, while testing the visual quality and performance on custom scenes gave us some insight into Lumen's capabilities and limitations, ultimately it may not be as indicative as taking a look at a real video game which implements it. The example I have chosen is *Fortnite*, since it comes directly from Epic Games, the company that should have the biggest expertise with Unreal Engine 5. Furthermore, Fortnite is often used to test the engine's upcoming features before they are publicly released [Gamg].

## 5.4 Testing Lumen in Fortnite

In this Section, I briefly analyze Fortnite's importance in developing Unreal Engine 5 and the reasons for which I chose it as my test object. Afterwards, I explain the testing I performed and describe the various configurations of graphical settings I used. Finally, I talk about the results and relate them to those from Sections 5.2 and 5.3.

### 5.4.1 Fortnite Introduction

Since its release in 2017, Fortnite is getting regular updates which not only add various dimensions of new content in the form of mechanics, gamemodes and micro-transactions, but also elevate its underlying technology. This ensures its very high and stable player counts, which are not expected to decline anytime soon [Bra24].

In 2022, Fortnite has been upgraded to Unreal Engine 5.1, which included the option to use both Nanite and Lumen. Incorporating these systems into Fortnite required some additional optimizations, which are described in Daniel Wright's article [WN23] and have heavily influenced Lumen's development.

However, the game's support has not stopped since then. In 2023, Epic Games released *UEFN*, which stands for *Unreal Editor for Fortnite*. This tool empowers the community with the ability to create fully customizable experiences that can be published directly in Fortnite. For more information on this technology, see the following article [Game].

In 2024, *Lego Fortnite* was introduced, serving as a beta-test for some of Unreal Engine's 5.4 new features, namely its Biome support for Procedural Content Generation and addition of GPU culling to hardware ray tracing, thus improving its performance. I assume that this trend will continue, and Epic Games will continue to use Fortnite as their platform for innovation and battle-testing Unreal Engine's upcoming features [Gamg, Unr24].

The game mode I chose for testing is *Fortnite Battle Royale*. It supports a day and night cycle, destructible environments, and both indoor and outdoor scenes, which makes it suitable for evaluating the visual quality of Lumen's

main features. For more details on this gamemode, please refer to Koczwara et al. [KCSm21].

These are only some of the reasons for which I believe Fortnite to be an excellent test subject, despite the fact that global illumination does not have a major impact on its gameplay. On the other hand, this allowed me to compare the Fortnite's performance and visual quality both with and without Lumen.

### 5.4.2 Fortnite Testing Description

I performed all tests using *PC 2*. For each configuration of the settings (shown in the next Section), I used software called *CapFrameX* [Cap] to measure the average CPU and GPU performance and FPS during a given time interval, along with other statistics. The specific interval I chose was 180 seconds, as that was enough time to test both large open world and smaller-scale indoor gameplay.

Note that having CapFrameX running in the background has certainly caused a slight ever-present decrease in overall performance, but the measured results for each configuration can still indicate the relative differences between them.

### 5.4.3 Fortnite Setting Configurations

The four different configurations of the graphical settings that I used during the testing are shown below. They are individually described in greater detail in the following Sections.

| | *Lumen off* | *Lumen High* | *Lumen Epic* |
|---|---|---|---|
| **Window Mode** | Fullscreen | Fullscreen | Fullscreen |
| **Resolution** | 1920x1080px | 1920x1080px | 1920x1080px |
| **VSync** | OFF | OFF | OFF |
| **Rendering Mode** | DIRECTX 12 | DIRECTX 12 | DIRECTX 12 |
| **Motion Blur** | OFF | OFF | OFF |
| **Aliasing and SR** | NVIDIA DLSS | NVIDIA DLSS | NVIDIA DLSS |
| **Nanite** | **OFF** | **ON** | **ON** |
| **Shadows** | High | High | High |
| **GI** | **OFF** | **Lumen High** | **Lumen Epic** |
| **Reflections** | **OFF** | **Lumen High** | **Lumen Epic** |
| **View Distance** | Epic | Epic | Epic |
| **Textures** | Epic | Epic | Epic |
| **Effects** | High | High | High |
| **Post Processing** | Epic | Epic | Epic |
| **HW Ray Tracing** | **OFF** | **OFF/ON** | **OFF/ON** |
| **NVIDIA Reflex LL** | ON + Boost | ON + Boost | ON + Boost |

**Table 5.2:** The various Fortnite setting configurations I tested. Note that **SR** stands for super-resolution and **LL** for low-latency.

Note that for all configurations, I used NVIDIA's DLSS 3.5, which renders the game natively at lower resolution and upscales it to full HD using a deep learning algorithm. While this slightly reduced the image quality by introducing a small amount of temporal flickering, I was mostly interested in Lumen's maximum potential performance, and according to NVIDIA, DLSS provides up to two times higher FPS [NVIb, NVIa].

#### Configuration 1 — Lumen off

This is the configuration I typically use when playing Fortnite, as it strikes a perfect balance between visual quality and performance, reaching 144 FPS in most situations. It renders shadows and roughly approximates global illumination by using ambient lighting, but does not produce any mirror reflections.

#### Configurations 2 and 3 — Lumen High and Epic

Lumen has two quality levels in Fortnite, which are called *Epic* and *High*. According to Wright's blog post [WN23], both of them support the same set of features, such as skylight leaking or auto-exposure, along with software and hardware ray tracing, but no differences between them are mentioned there. As of January 2025, the game does not give away any specific details either, and since Fortnite's source code is not publicly available, my best guess is that these quality levels directly correspond to those mentioned in Unreal Engine's performance guide [Gamb].

If that is the case, Lumen High traces only against the global distance field (described in Section 4.5.5) when using software ray tracing, while Lumen Epic traces against individual mesh distance fields (described in Section 4.5.2) first. The ray budgets for the final gather and reflections are also different. Lumen High uses 1/16 of a ray per pixel for the final gather and 1/4 for reflections, while Epic uses four times as much for both of these parts of the pipeline [Gamb].

Note that to enable Lumen in Fortnite, Nanite needs to be enabled as well, and either Unreal Engine's *TSR* or NVIDIA DLSS are strongly recommended to reach the target frame rates, which are 60 FPS for Lumen High and 30 FPS for Lumen Epic. The estimated frame-time budgets required to reach these frame rates are 4 ms and 8 ms respectively [Gamb].

Since these configurations support both ray tracing pipelines, I performed the testing first with software ray tracing, then with hardware ray tracing.

### 5.4.4 Lumen's Impact on Fortnite's Visual Quality

As already mentioned, I was mostly interested in testing Lumen's effect on Fortnite's performance. However, the impact it has on visual quality is what determines whether or not it is worth using in the first place. Therefore, I will first compare Fortnite's visuals with and without Lumen and analyze the performance afterwards.

Even when using the configuration without Lumen, Fortnite has very admirable visuals free of any immersion-breaking problems, with the exception of some LOD pop-ups. Due to its stylized graphics, realistic reflections and global illumination are not necessary for it to look visually appealing. Despite that, enabling Lumen made the game look significantly more realistic, especially in the indoor scenes, which are mostly lit by indirect lighting. The difference in quality is shown in Figure 5.29.


**Figure 5.29:** A visual comparison of Fortnite's visuals when using no global illumination and with Lumen enabled. Rendered on PC 2.

However, Lumen's global illumination had a negative impact on my gameplay experience. Specifically, when entering a building, there was always around a second or two long delay before indirect lighting fully lit the room, causing me to overlook all enemy players or traps hidden inside. This inconvenience was likely caused by either Fortnite's auto-exposure which is automatically used when enabling Lumen, or by the slow propagation of indirect diffuse bounces within the surface cache. Nonetheless, it felt like enabling Lumen offered a slight competitive disadvantage.

In addition, many of the indoor areas were plagued by a temporally unstable noise that was usually grouped at the corners of the rooms. Moreover, I occasionally encountered objects with completely black surfaces. This was presumably caused by missing surface cache coverage. I was also unable to tell the visual difference between the Lumen High and Lumen Epic settings, which may be only apparent during very dynamic situations, where slow light propagation is more noticeable.

The visual quality of software ray tracing also slightly differed from its hardware counterpart. First, the player character, which is a dynamically deformable mesh, was not properly shadowed and indirectly lit. Second, the

closing Storm, one of Fortnite's most important gameplay mechanics, was invisible in the distance. These differences are shown in Figure 5.30.


**Figure 5.30:** A visual comparison of Fortnite's outdoor scenery when using software (SWRT) and hardware ray tracing (HWRT). Notice the lack of direct shadowing on the player character and the missing storm when using SWRT. Rendered on PC 2 using the Lumen Epic configuration.

Finally, I was once again disappointed by Lumen's specular reflections, which are mainly found on water surfaces. Despite the fact that all reflections were slightly less noisy than during my custom testing (described in Section 5.2.5), there were still many view-dependent artifacts caused by Lumen's reliance on screen space ray tracing. One such artifact is shown in Figure 5.31, where the gun held by the player character partially occludes a small building near the lake. This causes the occluded part of the building to be missing in the reflection, despite physically being present in the game's world.

### 5.4.5 Lumen's Impact on Performance

In this Section, I present my conclusions on Fortnite's performance when using the aforementioned setting configurations. Figure 5.32 shows how the FPS of each configuration evolved over the tested time interval. Figure 5.32 shows their accumulated average and median FPS.

First, turning off Nanite and Lumen resulted in the highest and most consistent FPS overall. Specifically, it reached an average of 133.6 FPS, which is more than a 50 FPS difference when compared to the second-best configuration, that being Lumen High with software ray tracing.

Next, the difference between Lumen High and Lumen Epic was around 10 to 15 FPS, which is smaller than I expected. Despite this, I am not sure

**Figure 5.31:** A showcase of a visual artifact caused by Lumen's reliance on screen space ray tracing in Fortnite. Notice the partial lack of building in the reflection on the water surface. The character's gun was highlighted in red for better visual clarity, but no further modifications were made to the image. Rendered on PC 2 using the Lumen Epic configuration with disabled DLSS.

whether using Lumen Epic is worth it, since during my testing, I did not notice any improvements it had on the game's visual quality.

Since most objects in Fortnite's world (with the exception of terrain) are destructible, and the building mechanic utilizes a uniform grid, there is almost no object overlap. I assume that for this reason, enabling hardware ray tracing had very little negative impact on the average and median FPS. However, as evident in Figure 5.32, it introduced frequent stuttering, which was not present when using the software pipeline.

Compared to my old results from May 2024 (shown in Figures 5.34 and 5.35), most of the tested configurations were faster by up to 20 FPS and more stable, which I find to be a significant improvement that undoubtedly confirms Lumen's ability to run in real time when using the full HD resolution.

Overall, I recommend enabling Lumen to all Fortnite players who do not mind occasional visual artifacts or potential competitive disadvantages, as it has a very positive impact on the game's overall atmosphere.


**Figure 5.32:** A comparison of Fortnite's performance using 5 different setting configurations. The x-axis is limited only to the interval from 60 to 180 seconds to improve visual clarity. This Figure was created using CapFrameX and all data was measured in January 2025 on PC 2.

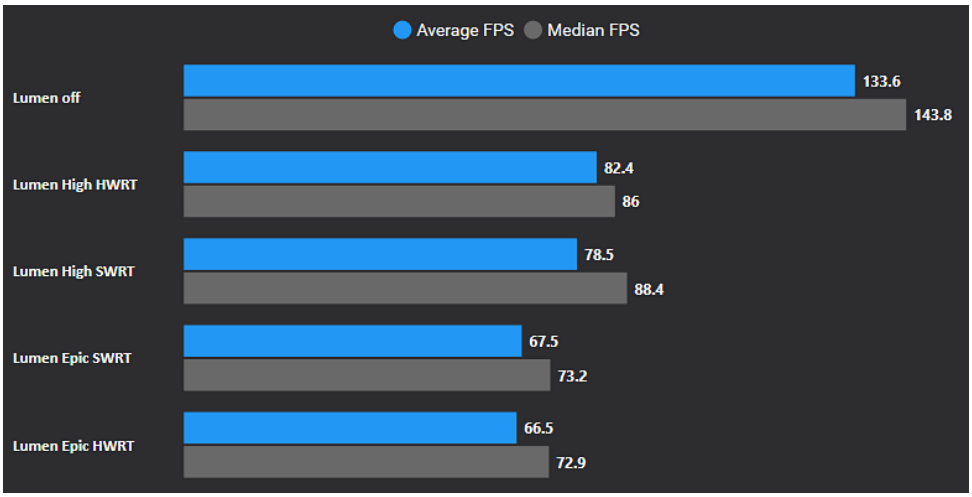



**Figure 5.33:** A comparison of Fortnite's average and median performance using 5 different setting configurations. This Figure was created using CapFrameX and all data was measured in January 2025 on PC 2.

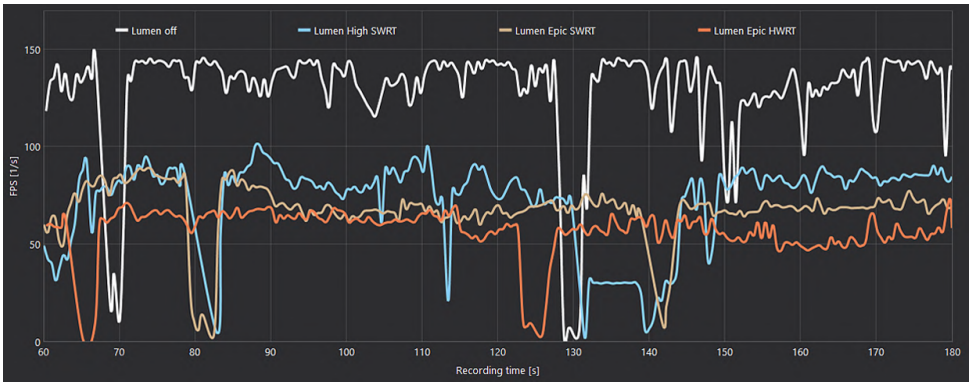



**Figure 5.34:** An older comparison of Fortnite's performance using 4 different setting configurations. The x-axis is limited only to the interval from 60 to 180 seconds to improve visual clarity. This Figure was created using CapFrameX and all data was measured in May 2024 on PC 2.

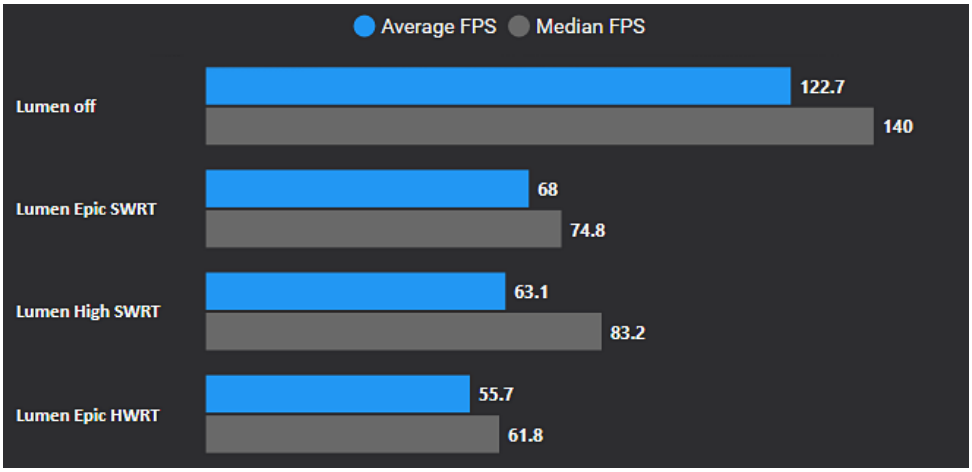



**Figure 5.35:** An older comparison of Fortnite's average and median performance using 4 different setting configurations. This Figure was created using CapFrameX and all data was measured in May 2024 on PC 2.

# Chapter 6

# Conclusion

In this final Chapter, I will first provide a very brief summary of this entire thesis. Next, I will share my overall opinion on Lumen, along with hinting at its future. Finally, I will mention a recent global illumination method that could potentially rival Lumen due to being conceptually more simple and innovative, and I personally find it much more promising.

## 6.1 Summary

Below is a summary of the key points I introduced, discussed, or analyzed in each Chapter of this thesis, along with references to the corresponding Sections, Figures, or Equations. Chapter 1 is omitted as it served only as an introduction.

- In Chapter 2, I discussed the role of *lighting* (2.4) in rendering and introduced related terms from computer graphics, such as *global illumination* (2.4.2), *BRDF* (2.4.3), *the rendering equation* (2.4.4), *Monte Carlo integration* (2.4.5), and *shadow types* (2.5.1). The summary of this Chapter can be found in Section 2.6.
- In Chapter 3, I used *Heckbert's light path expressions* (3.1) to describe and compare some of the illumination methods that are most widely used for both offline and real-time rendering, such as *ray tracing* (3.3), *Radiosity* (3.5), or *photon mapping* (3.6). I outlined their underlying concepts and hinted at their strengths and weaknesses. Furthermore, I discussed many of the most important *optimization techniques* (3.4) that are used to increase the performance of *real-time ray tracing* (3.4.5), as most of them are also utilized by Lumen. The summary of this Chapter can be found in Section 3.8.
- In Chapter 4, I described all *key components of Lumen* (4.3) and how they work together. Specifically, I described the differences between its *two ray tracing pipelines* (4.3.1), the way it utilizes *the surface cache* (4.7) to reduce the number of computations, and finally, its *final gather* (4.8) and *reflections* (4.9). I also mentioned *Nanite's impact* (4.7.2) on Lumen. The summary of this Chapter can be found in Section 4.11.

- Finally, I utilized all the information I learned during my research of Lumen to analyze its *visual quality* (5.2) and *performance* (5.3) using the three test scenes I created in Unreal Engine 5.5. Furthermore, I also played *Fortnite* (5.4) with various configurations of settings to better contextualize my following conclusions. A PDF file containing all the measured results can be found in the Appendix C under the name *Lumen Testing Results*.

## 6.2 Lumen and the Future of Ray Tracing

Given the ambitious goals Epic Games set out to reach with Lumen (explained in Section 4.1), I believe they mostly succeeded in achieving them. The technical details of Lumen, described in Chapter 4, are impressively thought out and innovative in its usage of the surface cache and screen space radiance caching. Furthermore, Lumen allows for a lot of configuration and customization, which I find to be both an advantage and a disadvantage, as setting it properly may require a lot of internal knowledge that requires some effort to obtain. This is underlined by the fact that most official sources describing Lumen have incomplete information and lack a proper, high-level summary of its components.

While my test scenes hinted at some of Lumen's unexpected visual shortcomings, namely the noticeable presence of temporally unstable low-frequency noise and flawed mirror reflections, its performance consistently reached the targeted 60 FPS, primarily when taking advantage of Nanite and Mega Lights. Moreover, when using high-end hardware, Lumen's performance was able to reach more than 120 FPS, especially with hardware ray tracing, which was significantly improved in Unreal Engine 5.5. However, all my tests were performed using full HD screen resolution only. Therefore, I cannot confidently say that Lumen's performance does not severely decline when using a higher resolution, such as 4K. This is an area that I personally wish to explore in the future once I get to own a suitable 4k monitor.

As of now, I do believe Lumen to be the most accessible and time-efficient dynamic global illumination option publicly available for artists working on projects aimed at running in real time. As time passes, the official documentation hopefully improves and its visual problems get ironed out.

Whether or not it is currently worth it for players to enable Lumen depends on the specific game, the experience they desire, and the hardware they own. I believe that Lumen's performance seems promising, and with the newly announced NVIDIA's RTX 50 *Blackwell* GPU series along with DLSS 4, it can soon be a non-issue instead of the deciding factor. However, if the performance increase provided by these new technologies is as dramatic as NVIDIA suggests, it is possible that using a simpler but slower and more visually consistent technique in Lumen's place, such as path tracing, may become more desirable [NVI25b, NVI25a].

Regarding Fortnite specifically, enabling Lumen did not improve my experience enough to justify the substantially longer render times.

## 6.3 Radiance Cascades

Developed by Alexander Sannikov for Path of Exile II, *Radiance Cascades* [San] is a real-time global illumination technique that builds upon Lumen's practical use of radiance caching without relying on spatiotemporal filtering. For this reason, it avoids many of the visual artifacts that can occur when using Lumen, such as ghosting or temporal instability.

The main idea behind Radiance Cascades is a formalization of the observation that as the distance between objects and the radiance probes increases, their angular resolution becomes more important than their spatial resolution. Therefore, instead of relying on only two levels of radiance cache (as Lumen does, with its screen space and world space radiance cache data structures), this technique utilizes a much larger hierarchy of probe atlases, where the first level has the smallest directional resolution and the largest spatial resolution. Each subsequent level increases the former and decreases the latter.

Furthermore, Sannikov claims that due to the aforementioned principle of this technique, the amount of rays required to trace each frame for full noise-free global illumination is constant and does not scale with the number of lights or objects in the scene, which I find to be an amazing achievement [San, Pat23, Sim24].

The effect this global illumination technique has on Path of Exile 2's visual quality is shown in Figure 6.1. When playing the game, I experienced no lighting-related visual artifacts whatsoever, even in highly dynamic situations. Unfortunately, the same cannot be said about Fortnite and Lumen.


**Figure 6.1:** A comparison of Path of Exile 2's visual quality with no global illumination and with radiance cascades GI.

# Appendix A

## Bibliography


[AA05] Hugues Hoppe Arul Asirvatham, *Chapter 2. Terrain Rendering Using GPU-Based Geometry Clipmaps*, 2005.

[AMHH18] Tomas Akenine-Mller, Eric Haines, and Naty Hoffman, *Real-time rendering, fourth edition*, 4th ed., A. K. Peters, Ltd., USA, 2018.

[ARBJ03] Sameer Agarwal, Ravi Ramamoorthi, Serge Belongie, and Henrik Wann Jensen, *Structured importance sampling of environment maps*, ACM Trans. Graph. **22** (2003), no. 3, 605–612.

[BMDS19] Steve Bako, Mark Meyer, Tony DeRose, and Pradeep Sen, *Offline deep importance sampling for monte carlo path tracing*, Computer Graphics Forum (Proceedings of Pacific Graphics 2019) **38** (2019), no. 7, 527–542.

[Bra23] Branch Education, *How do Video Game Graphics Work?*, 2023.

[Bra24] Thomas Branderhorst, *How Many People are Playing Fortnite in 2024?*, March 2024.

[Bur09] Magnus Burenius, *Real-Time Radiosity : Real-time global illumination of a static scene with dynamic lights using hierarchical radiosity*, 2009 (eng).

[Cap] CapFrameX, *CapFrameX*.

[Cin23] Carmen Cincotti, *The Rendering Pipeline / WebGPU / Video*, January 2023.

[Con24] Wikipedia Contributors, *Visible spectrum*, April 2024, Page Version ID: 1219133181.

[Cor22] Corridor Crew, *VFX Artist Explains the HARDEST Visual Effect to Make*, 2022.

[Dig24] Digital Foundry, *Tech Focus: TAA - Blessing Or Curse? Temporal Anti-Aliasing Deep Dive*, 2024.

[Dun16] Jesse Dunietz, *The Most Important Object In Computer Graphics History Is This Teapot*, February 2016.

[Epi24] Epic Games, *MegaLights in Unreal Engine | Unreal Engine 5.5 Documentation | Epic Developer Community*, 2024.

[Fel23] Petr Felkel, *Rendering and Illumination*, May 2023.

[Fla24] Market Flare, *(12) game engines market trends and analysis - opportunities and challenges for future growth (2024 - 2031) linkedin*, May 2024.

[FS22] Petr Felkel and Jaroslav Sloup, *Lighting and shading models*, March 2022.

[FvBS05] Petr Felkel, Jiří Žára, Bedřich Beneš, and Jiří Sochor, *Moderní počítačová grafika*, 2 ed., 2005 (cs-CZ).

[Gama] Epic Games, *Lumen Global Illumination and Reflections in Unreal Engine | Unreal Engine 5.5 Documentation | Epic Developer Community*.

[Gamb] ———, *Lumen Performance Guide For Unreal Engine | Unreal Engine 5.4 Documentation*.

[Gamc] ———, *Lumen Technical Details In Unreal Engine | Unreal Engine 5.4 Documentation*.

[Gamd] ———, *Mesh Distance Fields in Unreal Engine | Unreal Engine 5.5 Documentation | Epic Developer Community*.

[Game] ———, *UEFN - Getting Started*.

[Gamf] ———, *Unreal Engine 5*.

[Gamg] ———, *Unreal Engine 5.4 is here! Find out what's new.*

[Gamh] ———, *Volumetric clouds*.

[Gam23] ———, *Lumen Global Illumination and Reflections in Unreal Engine | Unreal Engine 5.3 Documentation | Epic Developer Community*, 2023.

[Gam24a] ———, *Gpu Lightmass Global Illumination In Unreal Engine | Unreal Engine 5.4 Documentation | Epic Developer Community*, 2024.

[Gam24b] ———, *Nanite Virtualized Geometry In Unreal Engine | Unreal Engine 5.4 Documentation*, 2024.

[Gam24c] ———, *Unreal Engine 5.5 is now available*, November 2024.

[GR21] Dimitria Electra Gatzia and Rex D. Ramsier, *Dimensionality, symmetry and the Inverse Square Law*, Notes and Records: the Royal Society Journal of the History of Science **75** (2021), no. 3, 333–348 (en).

[Gra15] Jens Gravesen, *The metric of colour space*, Graphical Models **82** (2015), 77–86 (en).

[Gre03] Robert Green, *Spherical harmonic lighting: The gritty details*, 2003.

[GTGB84] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile, *Modeling the interaction of light between diffuse surfaces*, Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques (New York, NY, USA), SIGGRAPH '84, Association for Computing Machinery, 1984, p. 213–222.

[Har95] John Hart, *Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces*, The Visual Computer **12** (1995).

[Hec90] Paul S. Heckbert, *Adaptive radiosity textures for bidirectional ray tracing*, SIGGRAPH Comput. Graph. **24** (1990), no. 4, 145–154.

[Jus20a] Justin Solomon, *Introduction to Computer Graphics (Lecture 16): Global illumination; irradiance/photon maps*, 2020.

[Jus20b] _____, *Introduction to Computer Graphics (Lecture 16): Global illumination; irradiance/photon maps*, 2020.

[Ju3] Dan Juříček, *Zobrazování detailních modelů v Unreal Engine* (en).

[Kaj86] James T. Kajiya, *The rendering equation*, Proceedings of the 13th annual conference on Computer graphics and interactive techniques, ACM, August 1986, pp. 143–150 (en).

[KCSm21] Michael Koczwara, Shailyn Cotten, Andrew Smith, and +1 6k moreUnlockedupdated, *Battle Royale Basics and Features - Fortnite Guide*, December 2021.

[KG09] Jaroslav Křivánek and Pascal Gautron, *Practical Global Illumination with Irradiance Caching*, Synthesis Lectures on Computer Graphics and Animation, Springer International Publishing, Cham, 2009 (en).

[Kim22a] J. J. Kim, *What Is Denoising?*, November 2022.

[Kim22b] _____, *What Is Direct and Indirect Lighting?*, August 2022.

[Kri] Jaroslav Krivanek, *Irradiance & radiance caching* (en).

[Lee21] Sugu Lee, *Screen Space Reflections : Implementation and optimization – Part 2 : HI-Z Tracing Method*, January 2021.

[May18] Benoit Mayaux, *2018 Mayaux - Horizon-Based Indirect Lighting (HBIL)*, 2018.

[Mic] Microsoft, *DirectX Raytracing (DXR) Functional Spec*.

[MSW21] Adam Marrs, Peter Shirley, and Ingo Wald (eds.), *Ray Tracing Gems II: Next Generation Real-Time Rendering with DXR, Vulkan, and OptiX*, Apress, Berkeley, CA, 2021 (en).

[Mut22] Mutual Information, *Importance Sampling*, 2022.

[NVIa] NVIDIA, *Deep Learning Super Sampling (DLSS)*.

[NVIb] _____, *NVIDIA DLSS Technology*.

[NVIc] _____, *NVIDIA OptiX™ AI-Accelerated Denoiser*.

[NVId] _____, *Nvidia turing gpu architecture*.

[NVI17] _____, *Spatiotemporal Variance-Guided Filtering: Real-Time Reconstruction for Path-Traced Global Illumination | Research*, July 2017.

[NVI20] NVIDIA Developer, *Ray Tracing Essentials Part 3: Ray Tracing Hardware*, 2020.

[NVI25a] NVIDIA, *DLSS 4 | New Multi Frame Gen & Everything Enhanced*, January 2025.

[NVI25b] _____, *GeForce RTX 50 Series GPUs and Laptops | Game Changer*, 2025.

[Pat23] Path of Exile, *ExileCon 2023 - Rendering Path of Exile 2*, 2023.

[PKTD08] Sylvain Paris, Pierre Kornprobst, JackTumblin Tumblin, and Fredo Durand, *Bilateral Filtering: Theory and Applications*, Foundations and Trends® in Computer Graphics and Vision **4** (2008), no. 1, 1–75 (en).

[Qui] Inigo Quilez, *Distance Functions*.

[rav20] raviramamoorthi, *Online Computer Graphics II: Rendering: Monte Carlo Path Tracing: Specific 2D Sampling Techniques - YouTube*, 2020.

[RDGK12] Tobias Ritschel, Carsten Dachsbacher, Thorsten Grosch, and Jan Kautz, *The State of the Art in Interactive Global Illumination*, Computer Graphics Forum **31** (2012), no. 1, 160–188, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2012.02093.x.

[Rea23] RealtimeZ, *Local Height Fog in Unreal 5 | Community tutorial*, August 2023.

[San] Alexander Sannikov, *RadianceCascades.pdf*.

[Sch24] Amanda Schade, *Unreal Engine 5.5 Preview - General / Announcements*, October 2024.

[Sed15] Kelly Sedor, *The Law of Large Numbers and its Applications* (en).

[Sha19] Pulkit Sharma, *An Introduction to K-Means Clustering*, August 2019.

[SIG21] SIGGRAPH Advances in Real-Time Rendering, *A Deep Dive into Nanite Virtualized Geometry*, 2021.

[Sim24] SimonDev, *Exploring a New Approach to Realistic Lighting: Radiance Cascades*, 2024.

[Sko23] Anna Skorobogatova, *Real-Time Global Illumination in Unreal Engine 5*, Ph.D. thesis, Masarykova univerzita, Fakulta informatiky, 2023.

[Sta15] Tomas Stachowiak, *Stochastic Screen-Space Reflections (SIGGRAPH 2015) - YouTube*, 2015.

[Sum23] Sum and Product, *Signed Distance Functions & Ray-Marching*, 2023.

[Tea22] Teadrinker, *English: Visualization of SDF ray marching algorithm*, February 2022.

[Tyc24] Jan Tychtl, *Webové aplikace běžící na straně klienta pro podporu výuky barevných modelů* (en), Accepted: 2024-06-18T14:31:31Z Publisher: České vysoké učení technické v Praze. Vypočetní a informační centrum.

[Unr21] Unreal Engine, *Lumen | Inside Unreal*, 2021.

[Unr24] Unreal Sensei, *Why Unreal Engine 5.4 is a Game Changer*, 2024.

[Ver21] Simon Verstraete, *Discussing the Possibilities and Drawbacks of Unreal Engine 5's Nanite*, June 2021.

[WJ05] Henrik Wann Jensen, *Realistic Image Synthesis Using Photon Mapping*, February 2005.

[WMLT07] Bruce Walter, Stephen R. Marschner, Hongsong Li, and Kenneth E. Torrance, *Microfacet models for refraction through rough surfaces*, Proceedings of the 18th Eurographics Conference on Rendering Techniques (Goslar, DEU), EGSR'07, Eurographics Association, 2007, p. 195–206.

[WN22] Daniel Wright and Krzysztof Narkowicz, *Unreal Engine 5 goes all-in on dynamic global illumination with Lumen*, May 2022.

[WN23] _____, *Lumen brings real-time global illumination to Fortnite Battle Royale Chapter 4*, 2023.

[WNK] Daniel Wright, Krzysztof Narkowicz, and Patrick Kelly, *SIGGRAPH 2022 Advances in Real-Time Rendering in Games : Lumen: Real-time Global Illumination in Unreal Engine 5*.

[Wri21] Daniel Wright, *Radiance Caching for Real-Time Global Illumination*, 2021.

# Appendix B
# Unreal Engine Scene Controls

Below are listed and explained the key-bindings for the Unreal Engine scenes I have created and used for testing Lumen.

## B.1 Feature Testing Scene Controls

This scene was used to showcase how Lumen handles different types of objects, lights, and mirror reflections with respect to their visual quality. Most of the objects in the scene and their special properties can be toggled in the user interface. For more details and results, please refer to Section 5.2.

- **Right Mouse Button** - toggles the cursor
- **W, A, S, D** - movement around $x$ and $y$ world coordinate axes
- **C** - switches the camera from 3rd person to 1st person and vice versa
- **R** - restarts the scene
- **Esc** - returns the player back to the scene selection menu
- **F** - toggles the frames per second counter
- **H** - toggles the user interface

## B.2 Performance Testing Scene Controls

This scene was used to measure the average GPU frame time Lumen takes to render a scene with configurations of objects and lights. All its variables, such as object and light counts, can be controlled through the user interface. For more details and results, please refer to Section 5.3.

- **Right Mouse Button** - toggles the cursor
- **W, A, S, D** - movement
- **Q, C** - vertical movement downwards

- **E** - vertical movement upwards
- **R** - restarts the scene
- **Esc** - returns the player back to the scene selection menu
- **F** - toggles the frames per second counter
- **H** - toggles the user interface
- **T** - performs the average frame-time test

## B.3 Overlap Testing Scene Controls

This scene was used to measure the average GPU frame time Lumen takes to render a scene with various counts of overlapping objects. This number can be controlled through the user interface. For more details and results, please refer to Section 5.3.4.

- **Right Mouse Button** - toggles the cursor
- **W, A, S, D** - movement
- **Q, C** - vertical movement downwards
- **E** - vertical movement upwards
- **R** - restarts the scene
- **Esc** - returns the player back to the scene selection menu
- **F** - toggles the frames per second counter
- **H** - toggles the user interface
- **T** - performs the average frame-time test

# Appendix C

## Attached files

Below you can find links to all attached files that I could not directly include as part of this document. This includes the built version of the Unreal Engine 5.5 project I created for testing, along with its source code and the *Lumen Testing Results* PDF document, which contains all the results I obtained during my Unreal Engine testing, along with a gallery of screenshots captured using the Feature Testing scene.

- Google drive folder, which includes all the files listed below.
- Source files of my test project created in Unreal Engine 5.5.
- Built version of my test project, packaged for Windows 10 and 11.
- Lumen Testing Results.
- Overleaf project