MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

Real-Time Global Illumination in Unreal Engine 5

Master's Thesis

ANNA SKOROBOGATOVA

Brno, Fall 2022

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Real-Time Global Illumination in Unreal Engine 5

Master's Thesis

ANNA SKOROBOGATOVA

Advisor: Mgr. Jiří Chmelík, Ph.D.

Department of Visual Computing

Brno, Fall 2022


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Anna Skorobogatova

Advisor: Mgr. Jiří Chmelík, Ph.D.

Abstract

Thesis is focused on the description of ray tracing techniques used in Unreal Engine 5 for dynamic global illumination. First, I describe the evolution of lighting methods to highlight features that are unique to ray tracing approaches. Then, I provide an overview of Unreal's lighting techniques, followed by a more comprehensive analysis of the Lumen's lighting pipeline. In the practical part, I summarize my experience of testing ray tracing methods inside an interactive application made in Unreal Engine 5.

Keywords

Computer Graphics, Global Illumination, Dynamic Lighting, Real-time, Rendering, PBR, Ray Tracing, Unreal Engine, Lumen, ...

Contents

List of Tables

List of Figures

Introduction

Games play a huge role in computer-based entertainment. The first titles like "Pac-Man" (1980) or "Space Invaders" (1978) were very simple simulations implemented as "monolithic" programs. Their game logic was hard-coded into the backend service code.

Nevertheless, starting with the late 1990s, more users could afford PCs, which created a larger demand for entertaining digital content. As computing powers of personal devices continued to grow, game enthusiasts became excited at the prospects of making more complex interactive systems. Games like "Quake"(1996), "Unreal"(1998) already rendered 3D environments and simulated some real life effects.

It was a big challenge, as one had to transform real-life observations into a set of abstract models describing physics, character navigation, decision making, user interaction. Furthermore, there were service-related processes to handle, such as rendering, streaming, IO processing, content management.

All this variety of functional parts convinced game developers to use modular architecture in their projects. In order to add new functionality into the gameplay creators only have to edit several modules rather than addressing the whole code base. Yet, as it was discovered in practice, some code chunks could be reused in multiple games, as they were not tied to the game logic. This practice led to further separation of the game-specific part from the context-free code. The latter becomes known as a "game engine".

> *"The line between a game and its engine is often blurry… We should probably reserve the term "game engine" for software that is extensible and can be used as the foundation for many different games without major modification."* (1, p. 11)

In the last 20 years, many game companies invested a considerable amount of time and effort developing engines for AAA games and high-level movie production: Unity (Unity Technolgoies), Unreal Engine (Epic Games), CryENGINE (Crytek), Frostbyte (EA), etc.

In this thesis, I would like to review one of the latest features introduced in Unreal Engine 5 (UE5) called *Lumen*. Presented as a new global illumination system, it adapts ray tracing methods for the needs of real-time applications in order to bring realism into dynamic scenes.

The goal of this thesis is to provide an overview of ray tracing methods used in Unreal Engine 5.0.3 for dynamic global illumination. I will start with outlining basic theory needed for understanding the scope and value of the rendering approaches. Then, I will proceed describing three ray tracing methods available in UE5 for dynamic rendering. In the practical part, I will describe my experience of testing ray tracing settings inside a small interactive application prepared in the Unreal Engine 5.0.3.

1 Human perception of Light

People perceive light in terms of lit and shadowed surfaces, from which they learn about the object's shape and its position in space. To reproduce the same effect in computer-generated scenes, researchers created several empirical models, whose main purpose was to mimic visual impact light makes on real environment.

Methods that belong to this class are *Lambert model*, *Phong reflectance model* and its altered version *Blinn-Phong model*. [2]

1.1 Empirical Lighting Models

The most famous representative of the empirical class is the **Phong Reflectance model**. It is known for its clear formulation of light behavior observed on solid surfaces. In its final version the Phong's lighting concept distinguishes between 3 shading components based on their reaction to the viewer's location and presence of light emitters in the scene. [3] Figure(1.1) shows Phong's additive lighting model.

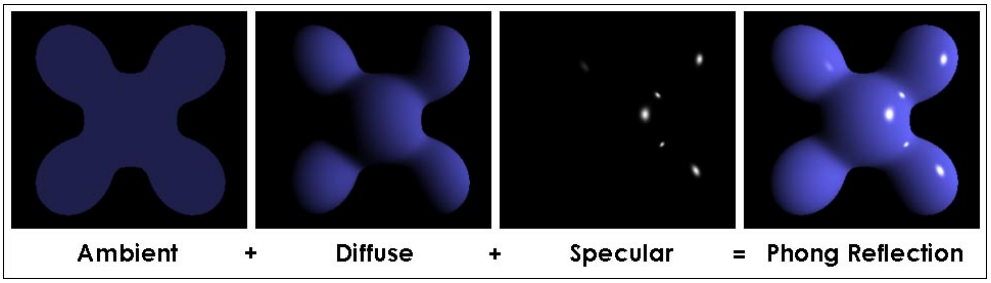

Figure 1.1: a) Components of Phong Reflectance model. [4]

Ambient is the base component that does not depend on any of the scene's object. It represents "minimal" uniformly scattered light that is always present in the scene and prevents objects from becoming black in unlit areas.

Diffuse component has the most significant visual impact on the scene, as it simulates the directional impact the light has on the shaded object. In simpler words, the more the surface "faces" the light source, the brighter it becomes.

However, the original concept of a diffuse falloff was borrowed from the *Lambert's* lighting method. For this reason, the diffuse component is often called *Lambertian lobe* or *Lambert's cosine law*. [5]

The intensity of the diffuse light is computed as a dot product of the surface normal $\overline{N}$ and the light direction vector L (left image in Figure (1.2)).

Specular component appears as a bright spot on the surface. It is sensitive to both light's and viewer's location. However, its size and intensity are driven through additional reflective property defined by the artist.

The specular light is computed as a dot product between the view direction $\overline{V}$ and reflected light direction $\overline{R}$ (right image in Figure (1.2)).

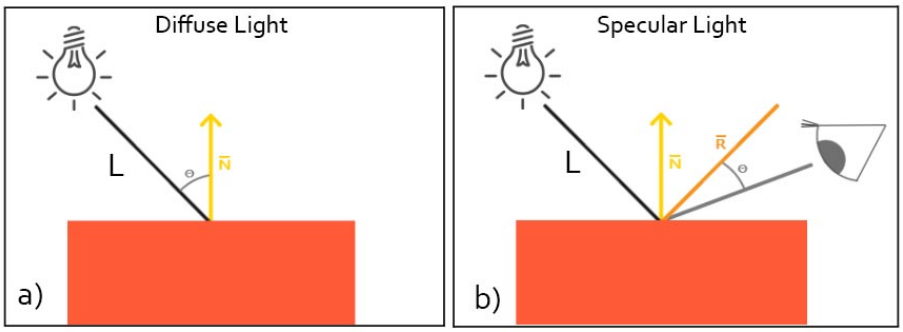

Figure 1.2: Computation of the Diffuse and Specular components [3]

For its simplicity, the Phong's model has been used in many 3D applications (i.e. 3Ds Max, 3Ds maya) as a fast shading method. Nevertheless, the biggest value of the *T.B.Phong's* research comes from his clear formulation of lighting terms, which are still widely used in the modern graphical research.

To summarize, implementation of empirical lighting models led to several important insights. However, every empirical model is still a very rough approximation of how light behaves in real world, which resulted in 2 major drawbacks:

1) The first one is the lack of limits for lighting and material values, since light components are not connected or limited by any mean. This makes artists fully responsible for feeding shader with the correct values in order to produce a realistic scene.

2) Yet, even if artists succeed in this task for one lighting scenario, the same input values would not necessarily work in other lighting setups. This would require either endless editing of input values in dynamic scenes or limiting the degree to which the scene lighting can change.

The best way to overcome mentioned difficulties is to use physically-based rendering (PBR) methods, which rely on a physical model of light spread in a real world. The PBR approach will be covered later in the 3rd chapter.

2 Physical concept of Light

> *"In physics, the term "light" may refer more broadly to electromagnetic radiation of any wavelength, whether visible or not… The primary properties of light are intensity, propagation direction, frequency or wavelength spectrum and polarization."* [4]

One might think of electromagnetic radiation as of a directed flow of massless particles called *"photons"*. They carry electromagnetic energy or*"intensity"* defined by photon's electromagnetic frequency or alternatively by its radiation *wavelength*. High level of photon's energy corresponds to high electromagnetic frequency or short radiation wavelength. The human eye perceives this form of energy as color. [6]

2.1 Light-Surface interaction

Light can be modeled as a beam of energy that keeps moving forward until it loses all of its energy due to collision with other particles.

So, as photons travel through space in one medium (air, water), they may eventually run into another medium represented by solid object. All objects are composed of materials whose physical properties are defined by their atomic-molecular structure. When photons arrive at the object's surface, they either collide with material particles or penetrate into spaces between them. In general, there are 3 main outcomes for when photon hits the surface: [7, 8]

- **"Energy absorption"**
  Photon penetrates an object, bounces among its particles, but never leaves the surface due to its low intensity value. The energy that has been lost to multiple collisions is then absorbed by material and converted into heat. This process is essential for the object's look, as materials absorb photons of certain wavelengths reflecting back colored light, which eye perceives as object's "true" color.

- **"Light Redirection"**
  Photon arrives at the surface, collides with material particles, but manages to leave the object at some point. This type of interaction, however, can be split into 4 different cases:

  - **Perfect Reflections**
    If surface is smooth enough, photon would bounce off at the same angle, at which it initially arrived, only on the opposite side of the hit point. On the macro level such light behavior creates highly reflective surface with "mirror-like" *Specular Reflections*. [7]

  - **Blurred Reflections**
    However, the majority of surfaces are not perfectly smooth, and would likely have some microscopic bumps and cavities that would slightly redirect arriving photons from perfectly reflected path. On the macro level that would be seen as *blurred Reflections*. [9]

  - **Diffuse Scattering**
    Developing the previous case: if the surface is relatively rough, meaning it has highly varying microscopic features, photons get reflected in random directions. As a result, some light particles may end up inside material, in which they travel for some distance before exiting at another surface point. Rough materials, therefore, have a certain amount of light scattering, which usually results in *matte-like Reflections*.[9]

  - **Subsurface Scattering**
    In the last case, if material physical properties allow longer travels inside an object, and its geometry is thin enough, photons may bounce all the way through the solid substance emerging from the onbject's back side. Such light behavior is called *Subsurface Scattering*, which gives the affected object a semi-translucent look. This effect can observed in materials like wax, milk, leaf, human skin. [10]

- **"Light Pass"**
  In case material has low scattering and absorption properties, photons can pass directly through the solid medium making object look transparent, like a glass. [7]

2.2 Radiometry theory

In the previous section I have described numerous cases of Light-Surface interaction and explained many visual effects from the point of physics. However, now we need to find a way to measure the strength of this interaction on macroscopic level. For this matter, physical science can offer several radiometric quantities that can be used to measure light over surfaces. (11, p. 267-272)

- **Radiant flux** ϕ
  quantifies the "amount" of light energy arriving at the surface. If we think of the Light as a function - a collective sum of emitted energy over multiple wavelengths (colors), the radiant flux ϕ will measure the total area of this function, returning a scalar value measured in Watts. Figure(2.1).

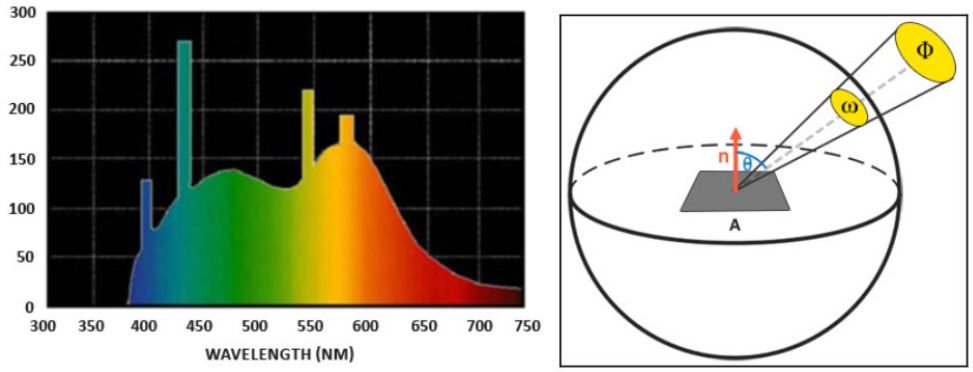

Figure 2.1: Radiometric measurements. [12]

- **Solid angle** ω
  defines area on a unit sphere taken by the projection of emitter's body onto the sphere. The area is expressed by the solid angle ω corresponding to the size of this projection. Figure(2.1).

- **Radiant intensity** I
  measures the amount of radiant flux per solid angle. It expresses the strength of a light source over a projected area on the unit sphere.
  $$I = \frac{d\phi}{d\omega}$$
- **Radiance** L
  quantifies the magnitude (strength) of the light arriving at the surface from a single direction. If expressed through another terms, radiance is the total amount of energy observed in area A over the solid angle ω and a radiant intensity I.
  $$L = \frac{d^2\phi}{dAd\omega * \cos\Theta}$$
  Additionally, light intensity must be scaled by $\cos\Theta$ which is a weakening factor, because according to *Lambertian cosine law* the light intensity must also depend on the angle at which it arrives at the surface.
- **Irradiance**
  is a sum of all radiances. It measures the total amount of light arriving at the surface from all directions.

2.3 Rendering equation

With the radiometric terms defined we can now measure the amount of light arriving at a surface point. For computation we shall use the Rendering equation created by *J.Kajiya* in 1986. [13]

The goal is to compute intensity of light leaving the point x. The equation considers 2 sources of light energy:

- the x point itself - in case it is a light emitter.
- all the incoming light cast from other emitters. The range of directions from which the point x can receive light is represented by hemisphere Ω.

$$L_0(x, \omega_0, \lambda, t) = L_e(x, \omega_0, \lambda, t) + \int_\Omega f_r(x, \omega_i, \omega_0, \lambda, t) L_i(x, \omega_i, \lambda, t)(\omega_i \cdot n), d\omega_i$$

Table 2.1: Notation for the Rendering equation [13]

$L_0(x, \omega_0, \lambda, t)$	total spectral radiance of wavelength λ that comes out from the point x along direction ω_0 at time t.
$L_e(x, \omega_0, \lambda, t)$	spectral radiance generated in point x and emitted from it.
$\int_\Omega ..., d\omega_i$	integral over the hemisphere Ω; represents irradiance - the sum of all incoming light.
$f_r(x, \omega_i, \omega_0, \lambda, t)$	***BRDF = Bidirectional Reflectance Distribution Function*** defines the amount of light reflected from ω_i to ω_0, based on the material properties.
$L_i(x, \omega_i, \lambda, t)$	spectral radiance of wavelength λ that comes towards the point x along direction ω_i at time t
$(\omega_i \cdot n) = cosi$	is a weakening factor of inward radiance, as it considers the angle at which the incoming light arrives at the surface. Is similar to the diffuse component in Phong lighting.

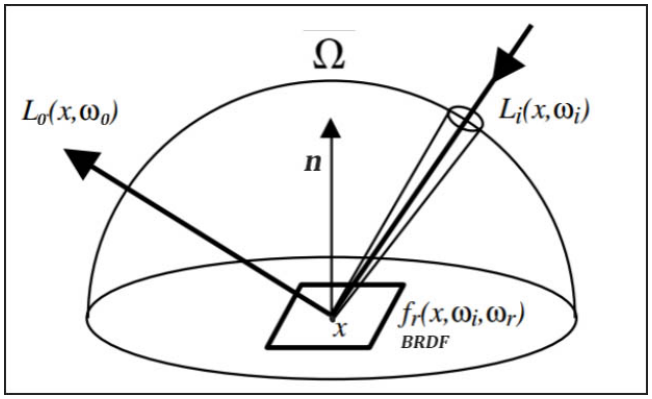

Figure 2.2: Variables used in the Rendering equation

All physically-based rendering techniques attempt to solve this equation to produce realistically-looking scenes.

One approach is based on *finite element methods* which is used in the ***radiosity algorithm***.

Perhaps, a better alternative is to *Monte Carlo aggregation* method included in many ray tracing algorithms, such as ***Path tracing*** or ***Photon mapping***.[13]

3 PBR: Physically-Based Rendering

The invention of the Rendering equation led to multiple attempts to bring physically-accurate lighting into actual 3D applications. Yet, the inability of computing machines to fully simulate physical light behavior by discrete mathematical operations led to certain compromises. For that reason, methods developed to support physical approach in rendering are called *"Physically-Based"* methods rather than simply *"Physical"*, as they were built to operate on simplified physical models. [12]

The first physical concept to be simplified was the set of radiometric measurements. Their scope must have been reduced to make them solvable by PBR approach:[12]

- The wavelengths in the ***radiant flux*** function were replaced by RGB color triplets.
- The ***solid angle*** has been reduced to an infinitely small number, limiting the incoming light to a single ray.
- The ***radiant intensity*** then began to correspond to the light emitted by a single point in space, which made it equal to the ***radiance term***.

The ***Light-Surface interaction*** model has undergone certain changes as well. Most of them are reflected in 3 core aspects of PBR theory, which is based on *Microfacet theory*, *Energy Conservation rule* and *BRDF* inputs. [7]

3.1 Microfacet theory

According to the original *"Light-Surface interaction"* definition, light is scattering over the surface due to the tiny bumps and cavities found at microscopic level on all physical objects. This misroscopic terrain forms the ***Microsurface***, which is represented in PBR by small-scaled little mirrors called ***microfacets***. [7, 8]

Each ***microfacet*** approximates a ***Microsurface*** region by averaging the surface normal in the defined area. However, evaluation of all the microfacets at once can be computationally intensive. So, the orientation of multiple microfacets is expressed with a cumulative statistical term called *"Roughness"* (*"Gloss"* or *"Smoothness"*). It defines the ratio of microfacets that face the same direction. The knowledge of the material's roughness value helps to approximate the proportion in which the incoming light reflects or scatters along the surface.

Figure (3.1) illustrates the impact the roughness value has on the size and intensity of the gloss spot. Smoother materials have a small, bright specular spot. Yet, when the roughness increases, the glossy area becomes wider and dimmer eventually converging on the Lambertian lobe. This example reflects the proportional change in the amount of reflected and scattered light rays.

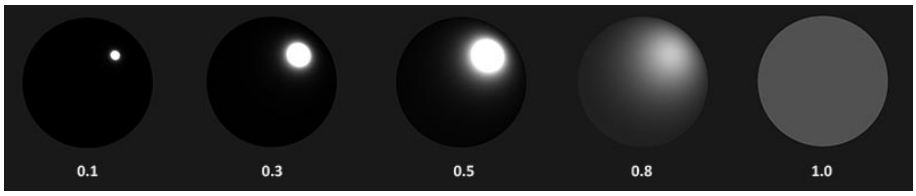

Figure 3.1: Dependence of light scattering factor on the roughness value.[7]

Yet, one more assumption about photon's behavior has to be made to keep the *Microfacet model* valid. As it was previously mentioned, the roughness parameter defines the scattering factor for a single point on the surface. However, it assumes that light always scatters in a small area around the sampled point and cannot perform long bounces. [7]

3.2 Energy Conservation rule

The concept of energy conservation states that
"an object can not reflect more light than it receives".[7]

The ***Light-Surface interaction*** model confirms this assumption by describing a number of outcomes, in which photons lose their energy in collisions with material particles.

The ***Energy Conservation rule***, however, recognizes only two cases, in which photon either penetrates the object's surface or bounces off at once. Both events are mutually exclusive, as for the light to be diffused, it must first penetrate the surface; thus, fail to reflect.

The opposite also holds, as shown in Figure(3.1), where a highly reflective object appears to have no color due to the lack of diffuse light.

The ***Energy Conservation rule***, therefore, binds the specular term to the diffuse component, establishing a strong connection between the Microsurface detail and material's reflectivity. Consequently, both shading terms can be controlled by a single roughness parameter, which affects both the size of the specular gloss and intensity of the diffuse color. [8]

3.3 BRDF

BRDF (*Bidirectional Reflectance Distribution Function*) evaluates the light at its collision point with an object and returns the amount of radiation reflected from the surface. [12, 8]

For the correct evaluation of the output BRDF shaders must load material properties through a number of input channels defined in a PBR texturing standard. The standard was developed to assist artists in the process of art creation and make their content look nice in physically-lit scenes. [10]

A traditional set of PBR channels supports textures like albedo, rouhgness, metalness, opacitym etc. delivering material-specific information to the shader. However, the PBR standard defines additional channels to support sophisticated lighting effects, such as *Subsurface Scattering* and *Ambient Occlusion* (*AO*).[9]

Subsurface scattering effect can be simulated by the shader, if it knows the object thickness at the sampled point. However, the only way to propagate geometry-related values into the shader is to upload a texture, which has this information "baked" into its texels.

The "baking" process is usually performed in 3D software like *Substance Painter*, which is capable of running complex analysis of geometric shape and projecting its results into 2D texture. For the

Subsurface Scattering effect the application generates a *thickness map* containing black values at thin parts of the model and a white areas for thick pieces.[9]Once this texture is loaded into shader, the rendering program will extract geometric-related information and shade the model accordingly.

In a similar way *Substance Painter* can produce *normal* and *displacement maps* for adding per-pixel geometric details to the surface.

Another example of a texture that affects local lighting is **Ambient Occlusion** (**AO**) map. This term represents an extra shadowing factor added to places that are partially occluded by surrounding geometry.

To summarize, **PBR** approach is a simplified version of a global physical process. It follows *energy conservation rule*, while approximating dense *microsurface* detail through statistical terms.

In order to compensate for technical limits at calculating sophisticated lighting features PBR shaders offer a large set of BRDF inputs to simulate local shading effects with precomputed textures.

One of the indisputable advantages of PBR approach is the standardization of the art content. All PBR assets can be easily shared among different PBR render engines, in which they look correct regardless of the environment's settings.

4 Ray tracing

"Ray tracing" is a generalized term for a large group of rendering methods, which evaluate lighting by casting rays into a scene and gathering values sampled at the hit points.

All methods in this group, despite sharing the same "tracing" concept, represent different approaches to evaluation of the global light.

Photon mapping [5], for example, casts rays from a light source and traces them throughout the scene by bouncing from surfaces. Yet, only those rays that arrive at the viewer's screen would contribute to the rendered image; others will be lost to off-screen and eventually terminated. So, the major drawback of photon mapping is a potentially high number of wasted rays, due which this approach is considered to be inefficient.

To manage the number of rays in a more reasonable way, *Arthur Appel* [5] suggested tracing in the opposite direction: by casting rays from pixels on the screen and tracing them into the scene, while searching for the light emitters. If one is found, its intensity value is then sampled and propagated back into the screen pixel. Yet again, only those rays that manage to discover light source along the path will contribute to the pixel's color. The problem, therefore, is similar to the one of the *Photon mapping*, but the number of wasted rays is usually much smaller.

For this reason, rendering an outdoor scene is considered to be a lesser challenge compared to an indoor space, since outdoor environments consist of many large areas exposed to the direct light (i.e sunlight), and typically do not need many light bounces to solve the shadowed areas.

The indoor scenes, on the other hand, receive light mostly from small local objects (windows, lamps, candles). Given a limited number of rays, the task of detecting a local light emitter in a room might be quite challenging. Yet, it becomes even more difficult, when indoor scene is filled with scattered light, as this can only be resolved through a long chain of ray bounces. [14]

Therefore, the challenges presented in the previous examples cannot not be solved without a further optimization of a ray tracing procedure. Based on the bouncing strategy, all ray tracing solutions can be classified to one of the 2 subclasses: *ray tracers* and *path tracers*. Although, these terms are often used interchangeably, there are substantial differences in the way they traverse a 3D scene. [15]

4.1 Ray tracers

The term *"Ray tracers"* originally referred to *"Conventional Ray Tracing"* methods, which were derived from one innovative technique introduced by *Turner Whitted* in 1979. [5]

The uniqueness of Whitted's approach was given by the use of tree data structure, which stored all bounces of a single ray computed before the rendering stage. As shown in the Figure(4.1), tree nodes represent hit points, while branches correspond to rays reflected from the hit surface.

This data structure was generated for every pixel on the screen. The generation process began with a single primary ray shot out into a scene, while searching for closest intersection with the scene geometry. Once the hit point was detected, it cast a so-called *"shadow ray"* into the known light emitters to check their visibility with the current hit point. If the shadow ray was not blocked by other objects and reached the light emitter, the hit point would sample its intensity value and propagate it back to the pixel.

Then, the whole process was repeated by generating more rays in the latest hit point and casting them further in random directions. The algorithm, therefore, was recursive, and would only stop after reaching a maximal number of hit points along each path.

The idea of storing ray paths in a spatial data structure, was quite a big breakthrough back in 1979, as it allowed to add more realism to the rendered scene by providing a tool to compute secondary lighting.

Whitted's approach, however, had one major drawback that came from its recursive nature. The number of rays generated for each layer of hit points grew in an exponential manner. This was considered to be inefficient, since the impact the consecutive light bounces have on the final image is typically smaller compared to the one made by the primary light.

> *"Due to the passivity of surfaces, it is widely known that the first generation rays as well as the light source rays are the most important in terms of variance that they contribute to the pixel integral. Second and higher generation rays contribute much less to the variance. But conventional ray tracing expends the vast bulk of the work on precisely those rays which contribute least to the variance of the image."* [14]

The need to spend the most time calculating the least significant part of the image led to the computational inefficiency in Whitted's approach and generated a request for further optimizations.

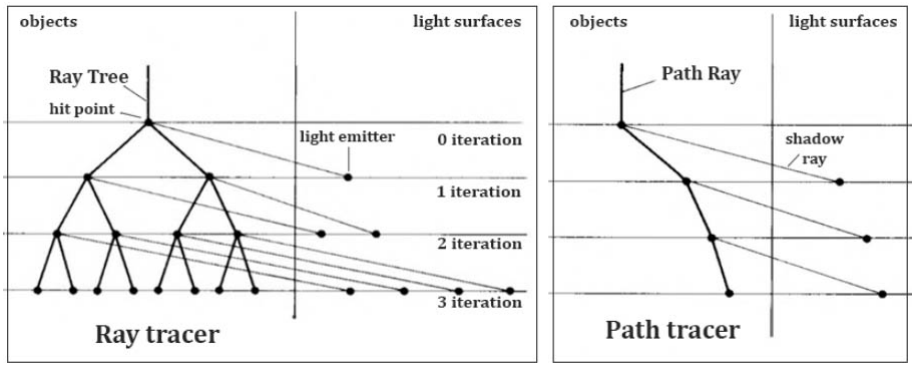

Figure 4.1: schematic of the Whitted ray tracer and Kajiya's path tracer. [16, p. 294]

4.2 Path tracers

The conventional ray tracing method described above can be called a brute-force solution, as it computes lighting in one rendering call by performing a broad traversal of a 3D scene. The computational costs of a ray-tracer approach, therefore, can be quite enormous.

In order to decrease the workload performed at a time, the original algorithm was altered by *J.Kajiya* in 1986 and became known as a *"path tracer"*. [14] The new version operated similar to the conventional ray-tracer method, but instead of branching multiple rays at each hit point, the path tracer followed only one branch, as it can be seen in Figure(4.1).

The path tracer, therefore, can be viewed as a ray tracer with branching ratio 1, meaning the number or processed rays remains the same throughout the whole computation.

However, the reduced number of ray samples could lead to the loss in visual quality of the rendered image. In order to prevent this, *Kajiya* additionally defined a new sampling strategy that would resolve only a limited number of rays at a time, but accumulate samples over multiple frames to approximate the final image.

The original idea of prolonging light evaluation over time belonged to *R.L.Cook*, who experimented with temporal sampling to produce blurred visual effects (i.e. motion blur) [17]. However, it was *J.Kajiya*, who firstly recognized the application of Monte Carlo-like methods in his approach.

> *"In 1984, Cook . . . , introduced distributed ray tracing. This approximation uses an extension of the three component Whitted model resulting in a more accurate scattering model. . . The innovation that made this possible was the use of monte carlo like techniques for the evaluation."* [14]

The Monte Carlos methods can be defined as:

> *"...a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results. The underlying concept is to use randomness to solve problems that might be deterministic in principle."* [18]

Thus, the alternative solution to a ray-tracer approach relies on an iterative process, which traces a few rays at a time and accumulates (integrates) the samples from previous iterations to approximate the final value. This process is known as *"Monte Carlo integration"*, and is an essential part of path tracing technique.

The original version of Monte Carlo approximation model relies on Markov chains, which make unbiased random choice of the direction, in which the generated ray will be cast into the scene. However, the correct choice of ray paths is important for path tracers, as the successful sampling strategy leads to fast discovery of light emitters and quicker convergence to the true result.

At the end of his groundbreaking work [14], J.Kajiya additionally described a set of biased sampling techniques, which provided a mathematical background for the choice of ray directions. The set was called **Hierarchical Sampling** and included several approaches, which pick the ray's direction based on the results of the previous sampling.

The *Adaptive sampling*, for example, tend to cast more rays from pixels that cover high contrast regions in the scene. Larger number of samples taken in a small area helps to increase lighting quality on the object's edges.

The *Importance sampling* method, on the other hand, while given a certain number of rays to resolve, shoots them in more "interesting" directions, along which the previous rays managed to sample high lighting values.

The biased sampling strategies, therefore, can considerably shorten the convergence time for the renderer. For that reason, they are widely used in modern ray tracing algorithms.

4.3 Hardware-Accelerated Ray Tracing

Getting back to the topic of computing illumination for dynamic game environments, it is worth noticing that researchers have been working on real-time rendering solutions for many years. Yet, among all explored strategies, ray tracing approach was considered to be the best candidate for the task, as its workflow resembles the way light spreads in the real world.

Ray tracing pipelines, therefore, could be automated to perform similar tests on all objects in the scene regardless of their type or relative position to the camera. Consequently, such methods would not require any supplemental data structures to solve the task, and probably would have few exception cases to be addressed separately in the code.

The attractiveness of the ray tracing approach, however, was undermined by its high computational costs, as electronic devices could not process sufficient number of rays to fully lighten the scene. For that reason, ray tracing techniques were used primarily in offline path tracers (i.e. V-Ray, Mental ray) and in a large film production, where it could take days to produce realistically-looking image sequences. [19, 20]

The real-time environments, nevertheless, could not be lit using offline rendering approaches, but required a distinct real-time solution instead. The one started to emerge, when *NVIDIA* provided hardware support for ray tracing operations on some of its high-end video cards in 2018.[21, 22] Following this event, the famous graphical vendor then announced a special API extension called **DXR** (DirectX Raytracing) for DirectX rendering interface.

DXR provides a set of functions that utilize the sources of ray tracing hardware. [23] The functions are called in the order corresponding to the unique rendering pipeline, which includes 5 additional shaders to offer larger control over the tracing procedures.

Generation shader, for example, defines of a ray casting strategy. The *Miss* shader specifies a color value for the unblocked ray.
The most important part of the pipeline, however, is a "hit group" represented by 3 shaders named *Intersection* shader, *Any-Hit* shader and *Closest-Hit* shader. Together, these 3 programmable blocks describe the ray marching strategy applied by the tracing process in its search for the intersection point with the scene geometry.[23]

The term *"ray march"* describes the iterative process of stepping along the ray and performing so-called *"hit-tests"* to detect the closest ray-blocking geometry.

The naive ray marching approach, for example, suggests making uniform, fixed-length steps along the ray and performing a hit-test at every stop. Yet, it has a drawback of skipping thin objects and performing a large number of redundant hit-tests.

In order to reduce the amount of wasteful computation, hardware ray tracing uses a special ray marching technique based on stepping through a simplified version of a scene described by **BVH** (Bounding Volume Hierarchy). [20]

According to the DXR specification, **BVH** is implemented as a two-level data structure with the ***Top Level Acceleration Structure*** (***TLAS***) covering the whole scene and multiple ***Bottom Level Acceleration Structures*** (***BLAS***) wrapping distinct meshes. The structures used in Unreal Engine are schematically shown in the Figure(4.2)

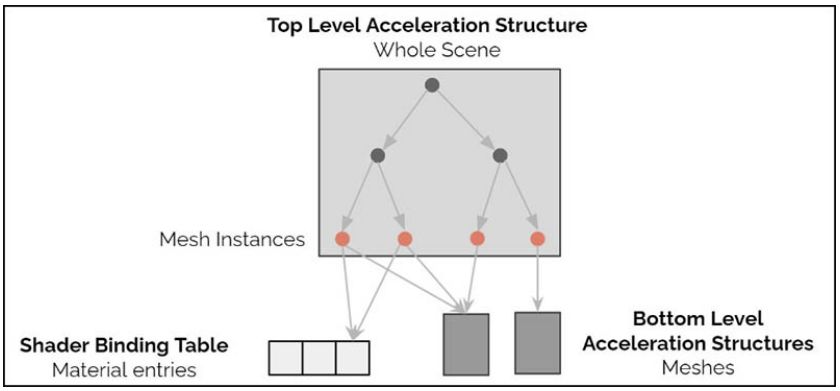

Figure 4.2: BVH acceleration structure used in Unreal's hardware ray tracing methods. [24]

Both levels are represented by their AABBs (Axis-Aligned Bounding Volumes), through which the ray is navigated to the mesh to perform an intersection test. It is also called a *"triangle-hit"* test, as it checks all triangles in the mesh to calculate intersection point using the triangle's barycentrics. In case such *"hit point"* is discovered, it it then stored along with its position and normal vectors, as well as material properties sampled from vertex attributes.

The accuracy of samples returned by triangle-hit tests is very high. Yet, their computational cost is proportional to the number of processed triangles. For that reason, the "triangle-hit" operation is implemented directly in GPU hardware, and its logic cannot be changed through DXR interface.

The tracing approach described above is relatively fast due to the quick switch between the BVH bounding volumes. Therefore, the BVH is an "acceleration structure" that significantly speeds up the ray tracing process. Its usage allows the fast discovery of blocking meshes, while the triangle-hit tests approximate the hit points on a higher precision level.

So, the technical improvements offered by DXR interface raised a wave of interest among developers, who were excited to use new ray tracing options inside their game engines.

5 Global Illumination in Unreal Engine 5

Unreal Engine has been widely known for its association with such famous game series as *"Unreal Tournament"*, *"Gears of Wars"* and *"Fortnite"*.(1, p. 27) After many years of development, the engine has turned into a powerful toolset enriched by features, developed for high-level products for EA, Ubisoft, Microsoft, and Disney.

One of the many reasons Unreal engine became famous for is its large collection of lighting options that make the lighting system quite responsive to the creator's input. Yet, the main difference between Unreal's lighting methods is the way they solve *direct* and *indirect* lighting components.

The **direct lighting**, also known as "primary lighting", can be observed on the surfaces that are directly "visible" (exposed) to light emitters in the scene. Due to its short light path, the primary lighting can be quickly evaluated for any object in the scene, either dynamic or static. Yet, areas occluded by surrounding geometry do not receive any light and are shaded black.

However, for the scene to look realistic, all its parts must be tied together by guiding light energy into distant corners. Shadowed areas, therefore, should receive some light from other lit surfaces, which would require the evaluation of secondary light bounces. Yet, their evaluation is quite expensive and hardly achievable at realtime.

For that reason, many methods have been developed to approximate the **indirect lighting** term, which is also referred to as "global illumination" for its essential role in producing highly realistic images.

The most traditional way of solving secondary lighting is precomputing it for static objects and storing the result into special textures called *lightmaps*.

In Unreal Engine, lightmaps are generated during the light baking process governed by the program called *Lightmass*. [25]

For objects to be properly shaded and baked into a lightmap, their polygons must be unwrapped according to the lightmap's creation guildelines to prevent the shadowing artifacts from baking into the tex-

ture. [26] However, this workflow puts an additional stress on artists, who have to prepare texture coordinates for multiple UV channels and check their state by launching a long light baking process.

Once the global lighting is finally baked, it is projected only on the static geometry, whereas dynamic objects are affected solely by the direct light. In order to project indirect lighting onto dynamic objects as well, the Epic's team implemented their own version of *Irradiance Field*. [27]

5.1 Irradiance Fields

Irradiance Field can be defined as a 3D grid of gathering points called *probes*. Each probe gathers indirect lighting values from surrounding static objects and projects those values onto any dynamic object that comes near it.

Unreal Engine has made 2 iterations on this method by creating *Indirect Lighting Cache* (*ICL*) and *Volumetric Lightmaps* (*VLM*).

Indirect Lighting Cache was Epic's first attempt to implement irradiance field in its engine. The whole ICL setup happens at the light baking stage, when Lightmass covers 3D scene with lighting probes and computes indirect lighting for them. When dynamic object enters the ICL volume, the lighting algorithm interpolates values from the closest lighting probes and applies a single shade to the whole mesh. This method cannot apply more samples, since the interpolation process runs on CPU, which has no access to mesh triangles.

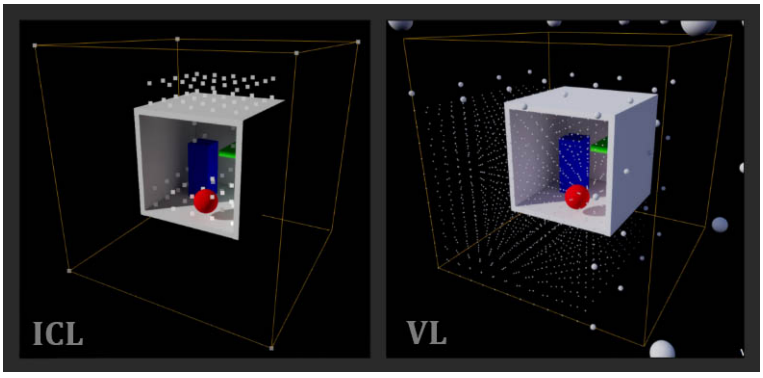

Figure 5.1: Probe placement inside the Lightmass Volume.[28, 29]

The probe placement in ICL grid is controlled by the bounds of *Lightmass Importance Volume*. By default, this volume contains a sparse sample grid, which may increase its density in areas, where characters would most likely appear; for example, in areas above the floor. Figure(5.1) shows an example of ICL probe placement inside the Lightmass volume.

However, the Epic's team decided to improve the quality of indirect lighting and replaced ICL with **Volumetric Lightmaps** (**VL**) [29]. The newer version offered 3 major improvements:

- The first one supported projection of multiple indirect samples onto a moving character, as the probe interpolation process has been moved to GPU.

- The second advancement changed the way the probes are generated inside the Lightmass volume. When the light baking process is launched, Lightmass program generates a sparse grid of 4x4x4 blocks, some of which are further subdivided into smaller blocks to provide better coverage for smaller static objects. This adaptive placement strategy results in more detailed and accurate lighting transfer to the moving objects. The right image in Figure(5.1) illustrates an example of VL probe placement in the scene.

- The third improvement turned a simple ICL sampling point into a VL spherical probe, which accounts for the direction of incoming light by storing it into a texel of its small cubemap.

Despite the advances brought by implementations of irradiance field, the scenes in Unreal still suffered from lighting artifacts. The most common one is the *"light leakage"* artifact. [27] It happens when 2 neighbor probes sample values from different lighting environments, and the interpolation process propagates one of the samples into the wrong environment.

The typical use case describes the "leakage" of outdoor sunlight into the dim house interior, as the interpolating algorithm fails to detect the house wall between 2 probes. This type of issues can only

be addressed by manual adjustment of probes position in the scene, which further increases the artistic workload.

Another major drawback of using irradiance fields is their dependence on the light precomputation stage, as it imposes significant limits on the content management and increases production time by the need of performing slow lighting updates.

Developers, therefore, continued their research on methods that could possibly compute the global illumination in real time.

5.2 Real-Time Ray Tracing

The announcement of DXR API extension in 2018 tempted many game developers into creating their own ray tracing solution for the high-end games. Epic's team was no exception, as it also showed interest in the new technology by iterating on their ray tracing implementation in order to meet the needs of newly tested products. [21]

The first results of DXR integration into the UE4 rendering environment were showcased at Developers Conference in 2018. The presentation highlighted the visual improvements in 2 demo movies named *"Reflections"* from ILMxLAB and *"Star Wars"* from Lucasfilm. [22]

The ray tracing algorithm developed during the production of these titles was considered to be a prototype, which selectively used some of the DXR functions inside the UE4 rasterization-based rendering workflow.

At the second iteration, though, developers fully switched to the DXR rendering pipeline in their attempt to achieve a stable performance with a complete set of ray traced features like ***Ray Traced Reflections*** (***RTR***) and ***Ray Traced Global Illumination*** (***RTGI***). In this form the method was then introduced in 2019 in UE4.22, where it was exposed to users as a *"Brute Force"* ray tracing method.

Nonetheless, the next milestone for developers was to improve the visual quality inside actual game environments, for which the *Fortnite* game was chosen as a testing platform [22]. Although, Fortnite was not the best candidate for its cartoonish, stylized graphics, the performed tests still provided a good feedback on the original "Brute Force" method uncovering its flaws. The algorithm, therefore, has to be changed in many aspects to become performant in dynamic scenes.

The goal was considered to be achieved, as some ray tracing features remained in Fortnite, while the altered version of the algorithm was included into UE4 2020 release as a further ray tracing method called the *"Final Gather"*.

5.2.1 "Brute Force" method

Before the Epic's team started working on a real-time ray tracing solution, it firstly implemented its "offline-version" running on the *NVIDIA* high-end hardware. Inside the engine this method is hidden behind the option called **"Path Tracer"**. It is only available as a view mode, since its output takes time to compute and is immediately invalidated by the camera move or a minor scene change. [30]

The indisputable advantage of the offline method, though, is the absence of performance limits; so, it can take time to render physically accurate images providing the reference for the developed real-time solutions.

The first ***Real-Time Ray Traced*** (***RTRT***) implementation is called **"Brute Force"**, which was designed to optimize the offline Path Tracer workflow to fulfill the requirements of real-time applications. [21]

Its rendering pipeline starts with the *Base-pass*, during which the scene is rasterized and rendered into G-Buffer. Then, the ray generation shaders fetch the information from framebuffers to get the direction, in which generated rays should reflect from the rendered surface.

In the first iteration of the algorithm, generated rays collected lighting samples from the probes of precomputed *Volumetric Lightmaps*.

In the second iteration, however, the algorithm computed global illumination using a more brute-force approach by doing path tracing with a secondary light bounces. The path length was defined by a user; yet, subsequent bounces could be terminated at any time by the Russian roulette process.

In the demos presented in 2018, the Epic's team used only 2 light bounces to keep performance under a certain limit, as every ray bounce invoked an expensive material sampling operation at every hit point. As the number of secondary rays grew, the brute-force algorithm could no longer deliver the result at interactive rates, and, therefore, was mainly used for pre-rendered cinematic clips.

5.2.2 "Final Gather" method

As it was previously mentioned, the usage of the *Brute Force* method considerably decreased the frame rates in Fortnite. [22] For that reason, the Brute Force solution has been modified to distribute its multi-bounce sampling over several frames.

The updated version has been named a **"Final Gather"**, as it collected shading values from a lighting buffer formed by gather points. Each gather point was implemented as a circular buffer that stored a sequence of samples acquired from secondary light bounces. Additionally, it had a record of a world space position of the screen pixel, for which the point gathered light samples.

The lighting pipeline for one frame consisted of 2 steps: sampling and gathering. At first, the algorithm executed the Brute Force procedure to evaluate one light bounce and add its output to the gather point's sequence. Then, the point's bounce chain had to be reconstructed to include only valid light samples, as the camera movement changed the pixel's position and, thus, the gather point's position in space. Such change inevitably invalidated some of the stored samples, and accumulation of multiple invalid samples could cause severe ghosting effects. In order to prevent those, the gathering point initiated the path reconnection procedure, which iterated over its stored samples and checked their visibility with the updated point. In the last step, the reconstructed light path was used to accumulate validated light samples and reproject them from the gather point into the screen pixel. [22]

In other words, *"Final Gather"* is an optimized version of the |textitBrute Force approach. It gains performance by time-slicing longer light paths and accumulating intermediate results in the gather points.

The visual quality of the image, produced by the *"Final Gather"*, might be worse compared to the one of the *Brute Force* approach, since the newer method struggles to keep its values valid for the processed frame. Nevertheless, limiting the number of processed light bounces per frame is what makes the Final Gather performant enough to be used in games. The *"Final Gather"* approach, therefore, can be considered the Unreal's first real-time ray tracing method.

Despite the certain wins gained through optimized shader workflow, both ray tracing methods could not be further improved due to the limited DXR support, which provided almost no control over the steps implemented in hardware.

For example, one of the major bottlenecks of hardware ray tracing was the updating speed of the BVH acceleration structure. The bounding volumes for the static meshes could be built just once with no further updates. The dynamic objects, however (i.e. skeletal meshes) needed local "refits" of BLAS volumes to fully cover the moving geometry. As it was discovered at *Fortnite's* testing, the update of the BVH was extremely slow in highly dynamic scenes, which caused a significant drop in framerates. [22]

Another drawback was the algorithm's inability to fastly process overlapping meshes, as BVH did not distinguish between object instances captured by a single BLAS volume. This would force the traced rays to check all the encapsulated meshes and their triangles inside that volume.

Considering the mentioned drawbacks, the Epic's team decided on moving to their software ray tracing solution. It was planned to make the new method utilize the engine's native data structures to accelerate the tracing process and overcome the limits imposed by hardware ray tracing API.

Therefore, the ray tracing methods described above were considered to be merely a stepping stone for the new lighting system designed as a replacement for older ray tracing versions. For that reason, both the *Brute Force* and *Final Gather* methods are defined as deprecated inside the engine.

5.3 Lumen

According to the Epic's documentation page:

"Lumen is Unreal Engine 5's fully dynamic global illumination and reflections system that is designed for next-generation consoles." [31]

Lumen, as a new global illumination system, was developed to overcome the limits of UE4 hardware-accelerated ray tracing methods.

Despite the high quality of ray traced outputs, RTGI pipelines could not be majorly improved due to the limited call base of the DXR API interface, and the DXR technology itself was not widely supported by the users' hardware.

For this reason, researchers from the Epic's team developed a software solution, which closely followed the hardware RT algorithm, but was highly optimized due to its usage of engine's native data structures. [32]

After a long series of tests, the primary version was further enriched by a large set of switchable options, which has turned the original method into a flexible and highly adaptable lighting pipeline known in UE5 as Lumen.

As of today, Lumen supports several ray tracing methods, for which it is often described as a *hybrid ray-tracing pipeline*. User, therefore, may choose between the software and hardware ray tracing modes based on the available hardware's capabilities.

- The **software lighting mode** is the Lumen's default setting supported by the widest range of hardware and platforms. However, it is limited in types of geometry and materials it can effectively work with.

- The **hardware lighting mode** is represented by a slightly modified version of the original RTGI algorithm known from UE4. It provides the best visual quality for the rendered image, but comes at a higher computational cost.

Similar to DXR algorithm, Lumen's ray tracing pipeline consists of 4 computational steps, which includes:

- detection of hit points using acceleration structures
- extracting surface properties at detected hit point
- evaluating the light spread in the scene, BRDF evaluation.
- propagating the computed samples back to screen pixels.

All the stages mentioned above will be further described in the next sections.

Since the software mode is considered to be main target for all of Lumen's innovative techniques, I will mainly focus on describing the

software lighting pipeline in the next sections.

Important note: *all the features covered in this chapter are valid for the engine's version Unreal Engine 5.0.3*

5.3.1 Acceleration Structures

Lumen's software solution is built upon a sequence of 3 ray tracing methods that differ in the size of area they cover and the precision of the scene traversing steps. Figure(5.2) shows a frame rendered by all three ray tracing methods.

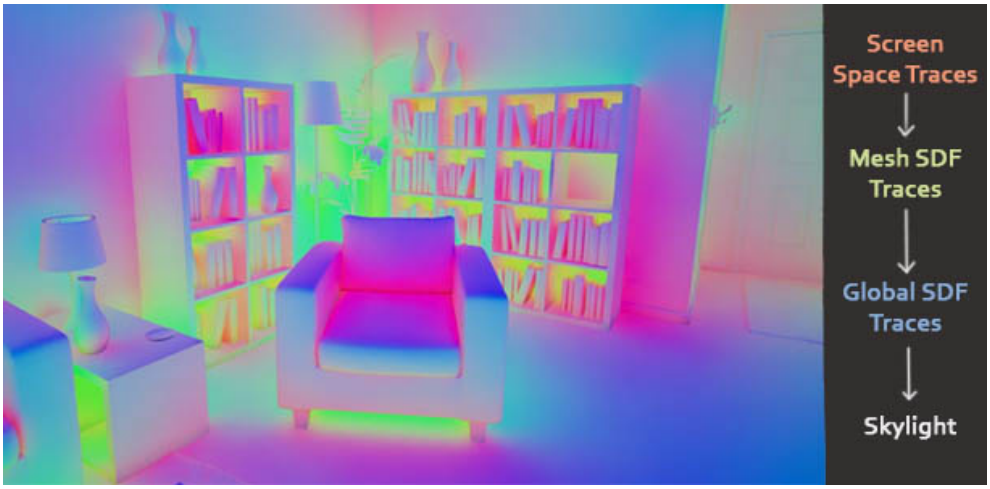

Figure 5.2: The frame is rendered through a combination of 3 ray tracing methods. [33]

As it was explained in the previous chapter, the speed of the scene traversal is affected by the type of used acceleration structure.

The first acceleration structure Lumen casts rays against is a **Hierarchical Z-Buffer** (**HZB**), as tracing in screen space is considered to be the most accurate method at short distances [33].

However, if rays do not hit anything in screen space, they are further tested against another acceleration structure called **Mesh Distance Fields** that cover all objects around the viewer. By missing those objects, each ray is then traced through a rougher scene representation

defined by **Global Distance Field**. Yet, if some ray misses all acceleration structures, it is automatically blocked by a global cubemap sphere called **Skylight**.

Hierarchical Z-Buffer

At first, rays are traced in screen space against the *Z-Buffer*, or alternatively - *depth buffer*. It is a 2D grayscale texture that stores depth values interpreted as distances between the viewer and the scene geometry. Knowing the distances to the rendered objects allows "reconstruction" of the visible 3D surface, against which the algorithm can trace.

However, tracing a scene on a per-pixel basis is quite inefficient. In order to speed up the ray marching process the algorithm creates a mipmap chain for the Z-Buffer turning it into ***Hierarchical Z-Buffer*** (***HZB***). [34]

A *"mipmap chain"* is a collection of downsampled textures called *"mips"*, where every *mip* has twice as lower resolution as the previous one. Figure (5.3)

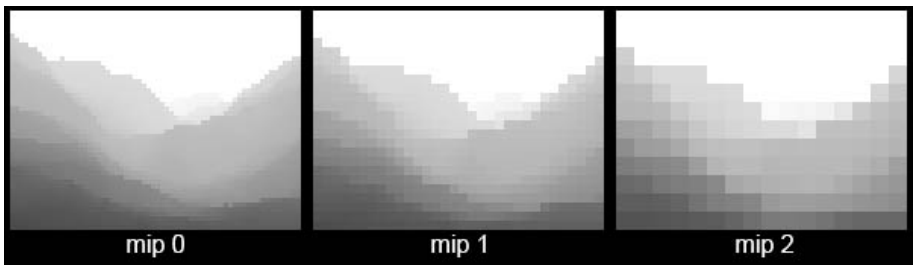

Figure 5.3: mipmap chain stored in Hierachical Z-Buffer. [35]

When a ray marches through the screen pixels, its depth value is compared to the depth value stored in the texture mip. If the ray does not intersect the surface, the active texture mip is replaced with a coarser image, and the ray is forced to skip a “larger” distance by jumping to the next texel. And in the opposite case: if ray hits the surface, the active mip is replaced with the finer image helping to approximate the potential hit point.

Figure(5.4) illustrates this process by showing texels as bluish columns whose height approximates the depth of the surface (white line). When the ray fails to hit the surface in the current texel, it jumps to the next one lying in a coarser mip; thus, taking a larger transition step and accelerating the ray march.

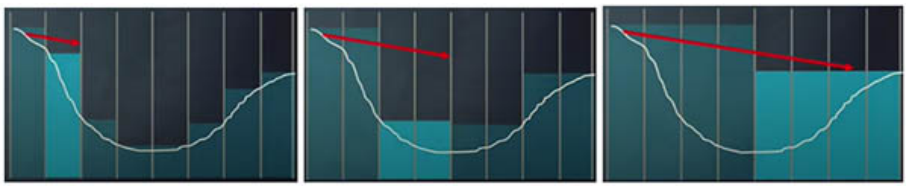

Figure 5.4: Ray marching through mips of Hierachical Z-Buffer.

As could be seen in the example above, ray marching in screen space is not much different from ray tracing in 3D space, as both methods share similar stepping problems.

The advantage of tracing scene in screen space is relatively accurate representation of geometry visible in camera view. This technique, therefore, proved to be helpful at covering mismatches produced by other less accurate tracing methods.

However, all screen space rendering methods have one major drawback, which is their inability to trace objects beyond the screen borders.

Distance Fields

Once the screen space tracing step is finished, the software mode continues to trace unresolved rays against another acceleration structure represented by *Mesh Distance Fields*.

Distance Field can be defined as a function that for any point in space returns a distance to the closest surface. Additionally, there exists a modified version called **Signed Distance Field** (**SDF**), which returns positive values for samples taken outside the object's volume and negative - for points sampled from inside of that volume. [36]

Finally, there is a **Mesh Distance Field** (**MDF**) implemented in Unreal Engine as a volumetric 3D texture that wraps the object's mesh. Its texels return both negative and positive values to be interpreted as distances to the object's surface. [37]

When a traced ray enters MDF volume and samples it at certain point, it receives a value by which the ray can safely skip the empty space to potentially arrive at the object's surface.

Due to the fact that *distance* is a scalar value, it can be viewed as a radius of a circular area that is safe to skip. For this reason, the process of ray marching through distance fields is also called ***sphere tracing***[36].

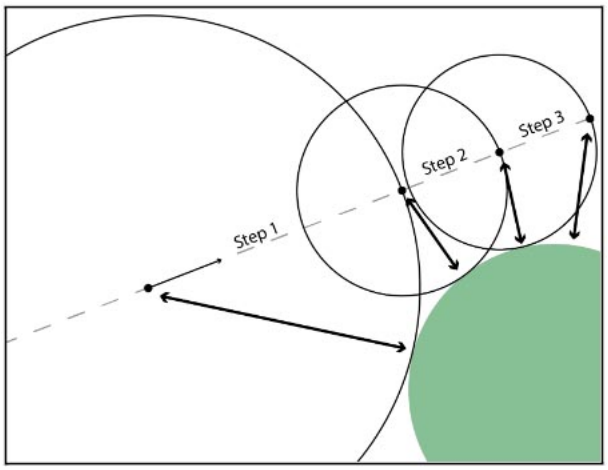

Figure 5.5: Ray Marching: Sphere Tracing. [15]

In brief, MDF is a ray-navigating texture that approximates the object's shape. Similar to 2D textures, it can be also streamed at runtime, appearing at a lower resolution when is far from the camera. For that reason, UE5 automatically creates a mip chain each time the MDF is generated, which usually happens during an object's import. [32]

When comparing MDF's quality reduction to mip switching in HZB, both methods appear to be similar in the way they accelerate the ray marching process.

By default, MDFs are only traced, when they are no farther than 40 meters from the camera, as even at the lowest resolution they cause too many hit tests near distant objects. So, beyond that range rays are tested against a rougher representation of a scene called **Global Distance Field**. Written as **GDF**, it is a global spatial structure created by merging all the MDFs into a set of 4 voxel grids called *clipmap*, stored in virtual volume textures (VVPs).

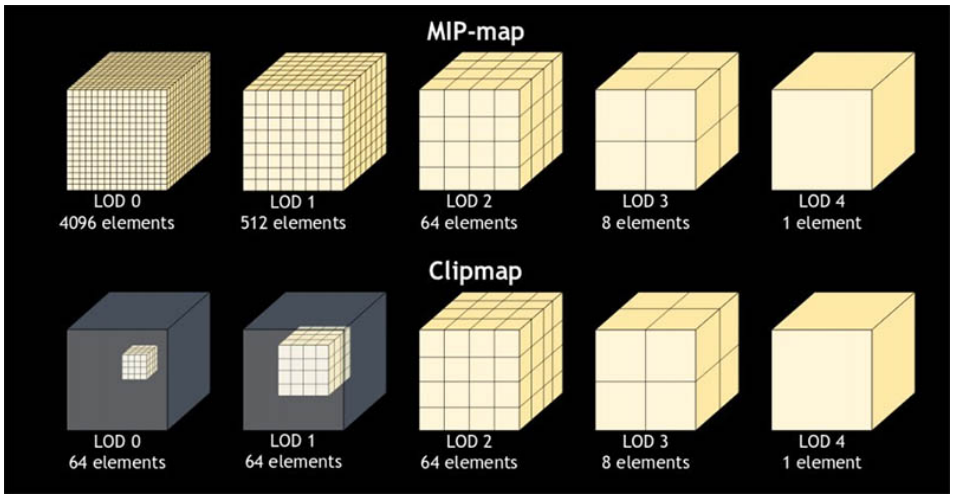

Figure 5.6: Mipmap vs. Clipmap. [38]

Clipmap can be described as a set of nested 3D grids centered around the camera. Each grid, also called a *level*, covers some area with a certain precision. [27, 38]

As shown in the Figure(5.6), the smallest grid around the camera point *LOD0* is the densest, whereas all other grids cover larger areas at a much lower resolution.

When tracing from the center of a GDF volume, the algorithm loops over all clipmaps starting from the smallest one. If the ray marches through the volume unblocked, it continues to be traced against the larger clipmaps until the hit point is found.

Therefore, GDF tracing scheme is very similar to the one of MDFs, as can be verified in Figure(5.6). Yet, instead of a MDF mipmap hierarchy GDF uses a clipmap hierarchy to simplify geometry in distant areas. Consequently, GDF supports very fast traces with a low number of hit tests in large areas, but lacks the accuracy of original MDF volumes.

Besides the multi-level spatial hierarchy, clipmaps are also known for their update caching strategy, which refreshes values only in places of scene change. [27]

An example is illustrated in Figure(5.7). As the camera moves aside, the clipmap grid also “shifts” along with it. Once the largest grid gets 4 new points on the right side, only their values are evaluated,

whereas the rest cell points reassign results known from previous computations.

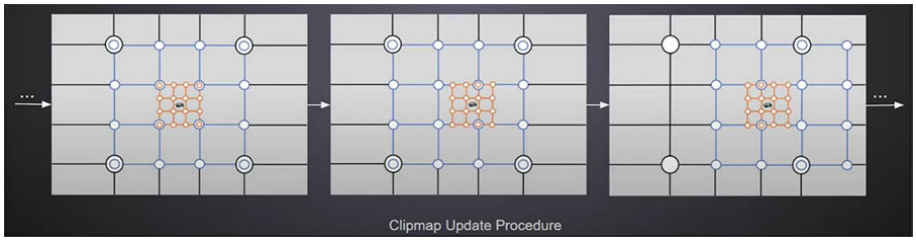

Figure 5.7: Clipmap update procedure. [27]

Based on the previous examples, it is safe to conclude that the switch between different acceleration structures happens because of the changing precision requirement based on the distance from the viewer.

However, distance fields cannot be used for shading calculations, as they do not store any vertex attributes from the original mesh. The only information to be received from hitting the approximated surface is the point's world-space position and its surface normal. Yet, there is no way to extract further surface properties for BRDF evaluation by analysing the hit point. [33]

5.3.2 Material Sampling

Distance fields exist primarily for fast collision tests, and do not store any surface inputs. So, another data structure has been introduced to keep track of the surface-related information including both material and lighting samples. This structure is called the **Surface Cache**, and is implemented as a large 4K atlas texture containing unwrapped planes with properties acquired from original triangle meshes [32]. One of such atlases is shown on the left side of Figure(5.8).

These planes are known as **Mesh Cards**. As can be seen on the right side of Figure(5.8), Mesh Cards are axis-aligned building blocks, which wrap the original mesh from all 6 sides to provide shading values for any ray that overlaps them. Implemented as 2D raster textures, Mesh Cards capture geometry with a higher precision than a 3D voxel grid.

Every Mesh Card texel approximates material values found on the object during its orthogonal projection on the Mesh Card's plane. The density of material samples stored in the texture depends on its raster resolution, which is derived from the object's size. The resolution, therefore, is fixed and cannot be changed at runtime, except for the standard mipmap switching at memory streaming. [32]

Material values, however, are not sampled just once and, in fact, are frequently updated, which forces regular execution of projecting and rasterizing operations. In this way, Mesh Cards support dynamic material changes, which increases accuracy of indirect lighting spread.

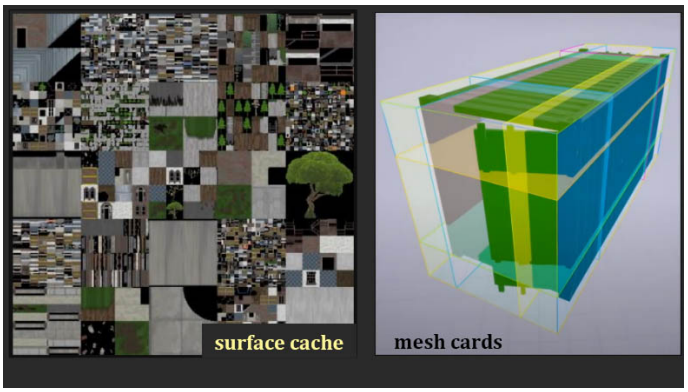

Figure 5.8: Surface Cache Atlas; Mesh Cards [33]

For better data coherency, all Mesh Cards are streamed inside the Surface cache atlas (Figure(5.8)). This makes the Surface cache one of the most frequently updated data structure in the Lumen's pipeline. Due to the large number of rewrite requests sent in one frame, the update of Surface Cache is often seen as a bottleneck in lighting computations. The main predicament for this process is a slow projection of raw material samples onto Mesh Card's texels, performed by a GPU rasterizer. It happens, because hardware rasterizers are designed to perform only one task at a time, putting multiple requests into a queue list, which increases the update time. [33]

Nanite Optimization

Fortunately, UE5 comes with another prominent feature called **Nanite** that significantly speeds up both the rendering process and material captures.

According to Epic's documentation page, Nanite is a *"virtualized geometry system to achieve pixel scale detail and high object counts."* [39]. In simpler words, Nanite is a new streaming method for 3D models, which loads only those polygons in a triangle mesh that are needed for rendering of the current frame.

Nanite is a replacement to the old LOD (Level Of Detail) *"geometry mipmap"* system, as it is capable of rendering hundreds of high polygonal meshes at any distance in real-time. In order to make Nanite work along the other engine's data structures, developers had to rewrite the whole rendering pipeline turning it into a GPU driven workflow. Furthermore, ts final implementation included several software rasterizers capable of performing multiple interpolation tasks in parallel (multi-view rendering). [33, 32]

Nanite's rendering mode, therefore, can significantly speed up the Surface Cache updates by supporting parallel Mesh Card's recaptures. For this reason, it is recommended to activate Nanite mode when using Lumen lighting system, even if final scene contains only low poly models.

Before moving to the next section, there is one more term that requires explanation. That is a **LumenScene**, which is considered to be Lumen's most important scene representation, as it features the most important scene components needed for the light evaluation. The first one is a hit acceleration structure represented by distance field shapes, while the second one is surface caching data, whose shading values are applied to those distance field shapes.

5.3.3 Light Accumulation

The process of light evaluation in a LumenScene can be split in 2 stages: [33, 38]

- computing direct lighting with a single bounce in screen space.
- adding indirect lighting by sampling volumetric lighting buffers that exist in 3D scene.

Direct Lighting

At the first stage, the visible part of a scene is rasterized into a set of 2D framebuffers. Each framebuffer stores a specific type of scene samples (i.e. albedo, normal, distance) required for the accurate geometry analysis and fast pixel shading.

Samples from the *depth* and *normal* buffers, for example, are needed to perform fast visibility tests between the rasterized geometry and mathematically defined light emitters.

In case the path between the surface and light source is not blocked, the surface should be lit, and the pixel covering this area may apply samples from the color framebuffers. Yet, if the tested surface is not on the emitter's line of sight, it is "occluded" by another object and lies in its shadow. Pixels at the shadowed areas are painted black, which gives a sharp contrast look to the rendered image, as can be seen in Figure (5.9).

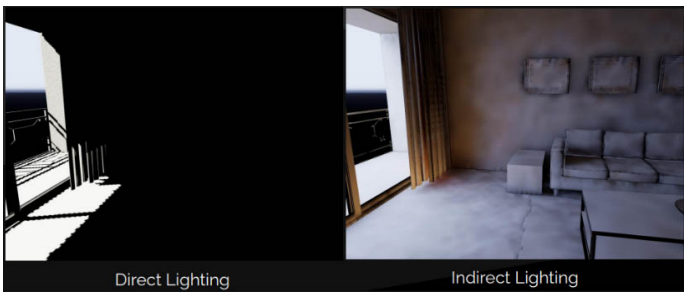

Figure 5.9: Direct and Indirect lighting components [33]

Indirect Lighting

So, direct lighting is relatively easy to compute, since it evaluates only one bounce between a light emitter and the pixel on the screen.

The indirect lighting, however, requires far more bounces around the scene to propagate reflected light into unresolved shadowed areas. Yet, such complex calculations cannot be performed in screen space, because framebuffers do not store multi-layer values for overlapping objects and cannot resolve bounces outside the rendered area.

For that reason, Lumen implements 2 spatial lighting buffers responsible for light propagation in the scene.

Indirect Lighting Probes

At the closer ray hits, global illumination is sampled from the Surface cache. The indirect lighting component is approximated using 2D grids of lighting probes generated on top of Mesh Cards. The density of probe grids depends on the Card's resolution used in Surface Cache. Figure(5.10) shows a scene covered with Mesh Cards' lighting probes.

Figure 5.10: Scene covered with indirect lighting probes. [33, 38]

Each lighting probe, supposedly a hemisphere, is unwrapped as an octahedral over the texel block of 4x4 resolution. (right schematic on Figure(5.10)). Texels encode directions in which the rays are shot.

In an ideal case, the indirect probe would trace dozens of rays in multiple directions to properly sample the environment. Yet, due severe performance constraints only a subset of indirect probes are allowed to cast a ray in one frame. The multi-bounce effect, therefore, is achieved by accumulating light samples over time. Due to the high cost of ray trace operation , the probes' locations and trace directions slightly jitter every frame in order to widen the sampling range.

Furthermore, the direction of every generated ray depends on the value of previous samples, which can lead to potential light emitters. For this reason, ray casting from indirect probes is considered to be *"feedback-based"*. [33]

However, despite all efforts, the values received by probes can be quite noisy, since the sampling rate is very low. The noise can be reduced by a so-called *"spatial and temporal reuse"*.

The spatial reuse is based on the image filtering technique, which performs bilinear interpolation of 4 indirect probes to compute a lighting value for the underlying mesh card's texel.

The interpolated value is then temporarily stored in the **indirect lighting atlas**, where it will be accumulated with previous light samples. The temporal accumulation is limited by 4 frames, after which the sample gets discarded. This period was made short to reduce ghosting effects during camera movements.

Once the light values are sufficiently approximated, they get injected into the mesh card's texel, where their intensity will be further affected by material properties during the BRDF evaluation. [33]

Volumetric Clipmaps

Mesh Cards proved to be a responsive surface buffer for MDFs traces. However, when it comes to evaluation of distant objects, Surface cache samples cannot be directly applied to GDF hit points, since GDF is a merged version of scene geometry with no records of distinct meshes.

So, in order to solve the global case, developers implemented a lighting buffer inside already mentioned clipmaps. Each clipmap's cell is then represented by a lighting voxel. The example of shaded voxelized clipmap is shown in Figure (5.11).

Figure 5.11: Volumetric Clipmap lighting, [33]

Each voxel collects material properties by merging samples from local Mesh Cards and gathering radiance values along each of its 6 axis-aligned directions.

Voxel, therefore, can be viewed as a cube of 6 Mesh Cards of a very low resolution, since every voxel side stores only one shading value. When a GDF hit point is detected, it gets a shading sample by interpolating 3 voxel sides aligned with a point's surface normal.

Reflections

The secondary light bounce, accumulated in sparsely distributed gathering probes, seems to provide "sufficient" amount of detail for indirect lighting component. That measure of "sufficiency" is what makes the software ray tracing applicable at runtime, as the balance between the visual quality and tracing speed cannot be found without accepting certain compromises.

As it was explained in the previous sections, Lumen tends to reduce the data accuracy almost at every stage in exchange for additional performance. At the same time, the system proves to be quite successful at hiding low resolution data from its user. Nonetheless, there is one component, in which software ray tracing method cannot cover its inner data loss.

Once the diffuse lighting is evaluated, the tracing algorithm begins to compute reflections, for which it has to sample the finest representation of the scene it has access to.

In software mode that would be a *LumenScene* represented by MDFs and Surface Cache. So, it does not provide any high-frequency details for mirror-like reflections.

In order to accurately shade the smooth surfaces the program needs the hardware ray tracing pipeline capable of performing traces against the actual mesh triangles. In Lumen's context, however, the hardware solution can work in 2 reflection modes: [32]

- **"Surface-Cache"** reflection mode
  picks shading information from the Surface Cache and projects its samples from Mesh Cards onto triangle mesh. So, both direct and indirect lighting are computed in a single evaluation pass.

- **"Hit-Lighting"** reflection mode
  represents a modified version of the original DXR reflection algorithm, which performs accurate hit tests for calculation of direct lighting. The material and indirect lighting values, however, are still sampled from the Surface Cache.

5.3.4 Final Gather

Final Gather is the name for the process of propagating light samples from *LumenScene* to the pixels on the screen.

Once the indirect lighting is accumulated in spatial light buffers, it has to be retrieved from the scene and transferred back to the viewer. Both the sampling and transfer tasks are solved by tracing rays from irradiance gathering points. Yet, similar to other Lumen's tracing stages, Final Gather separates the scene objects into distance ranges and solves them with different techniques. [40, 33]

Screen Space Radiance Cache (SSRC)

The first irradiance gathering structure is called **Screen Space Radiance Cache (SSRC)**. It consists of **Screen Space Probes (SSP)** that collect light samples by shooting rays from surfaces visible to the camera.

The probes are sparsely placed on the viewer's screen, as one SSP approximates a 16x16 pixel block instead of a single pixel. This allows to redistribute the ray budget, which originally supported only half a ray per pixel. The pixel block, however, gives the gathering point 32 rays to cast. [40]

It is often said that screen probes are placed directly on the screen plane to ensure the same density of samples every frame. It is true, however, the screen probe firstly shoots a ray to hit the primary (visible) geometry rendered at the pixel block, and only from that hit point it further casts those 32 rays to sample Surface cache values from surrounding geometry [38, 41].

As all 32 samples backtrack to the gathering point, their values are recorded into SSRC probe atlas, summed up and projected back to the pixel block by a single ray.

Based on this light gathering scheme, Lumen can be described as a ray tracer, whose multi-branch sampling is implemented via Surface Cache update model. At the same time, Lumen also displays features of a path tracer, as it accumulates light over time and implements certain sampling techniques to efficiently guide the sampling rays in space.

The first technique is called *adaptive sampling*, which suggests altering the probes' positions to support tracing with minimal number of probes. By default, SSPs operate at 1/16th screen resolution, which is quite low. However, by slightly jittering the SPP placement each frame and accumulating results over time, Final Gather provides a good sampling coverage for the pixel blocks, as can be seen in Figure(5.12)

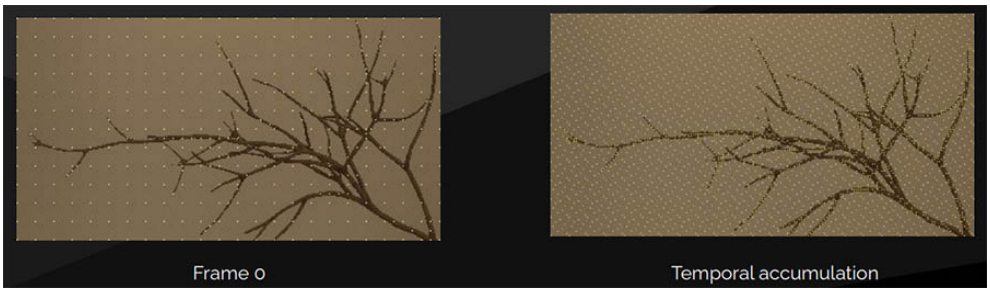

Figure 5.12: Adaptive generation of SSR samples [40]

Despite the growing number of sampled positions, the result would be very noisy. To mitigate this, the accumulated samples are spatially filtered (blurred) in the probe space, as possible loss of high-frequency information does not seem to be crucial for indirect diffuse component.

The filtering operation then provides interpolation values for all pixels on the screen. In case of an artifact that might occur in high contrast screen areas, Final Gather generates further screen probes to better approximate the problematic region. This concept is denoted as a *"hierarchical refinement"* of the grid. [14]

Regarding the implementation, all SSPs record their directed samples into the intermediate light buffer called **Radiance Cache**. This buffer stores a large texture atlas, in which every SSP receives a block of 8x8 texels. The screen probe is then unwrapped into this block using octahedral UV layout with texels corresponding to the sampling ray directions. At runtime, every texel stores sampled radiance value and a distance to the hit point.

To further improve the ray casting strategy, Final Gather implements *importance sampling* of the incoming light (Figure(5.13)). By treating the last frame's radiance and BRDF samples as an estimate

of irradiance, the sampling algorithm can redirect some of the culled rays into more prospective directions to speed up the convergence to the final result.

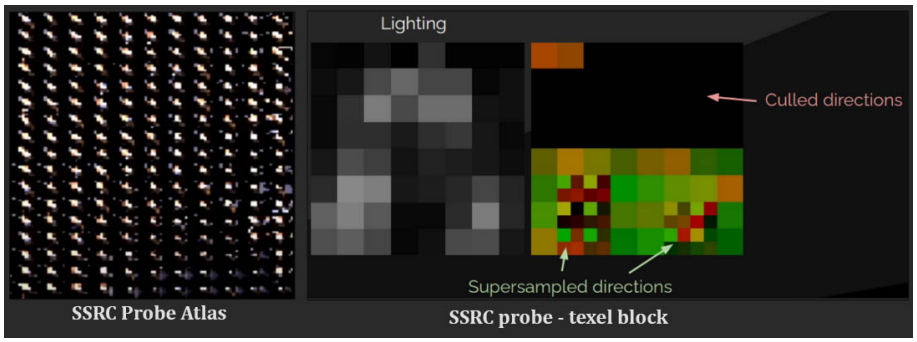

Figure 5.13: Left: SSRC atlas. Right: Importance sampling. [33]

World Space Radiance Cache (WSRC)

While SSRC is used for short ray traces, the distant lighting is solved with a separate sampling scheme called **World Space Radiance Cache** (**WSRC**). Like the name implies, the **World Space Probes** (**WSP**) originate at grid points of 3D clipmaps. Their spatial resolution is much lower compared to the one of the SSRC. However, the directional resolution of WSP is significantly higher, as each space probe takes a 32x32 texel block in its Radiance atlas. [40]

The **WSRC** was introduced to support stable distant lighting, as the directional resolution of screen probes might not be high enough to find small, distant light emitters in the scene. The clipmap distribution, therefore, was chosen to maintain a bounded screen distance between the probes to avoid over-sampling or under-sampling.

Unlike SSRC probes, WSRC probes have persistent allocations and survive across the frames. Their values are sampled in a moment when a casted ray from a screen probe gets farther than 2m from the screen. However, if the screen probe's ray does not hit anything in world space, it then samples the external *Skylight's* value.

As it was described above, both the SSRC and WSRC operate in downsampled resolution space, while their sampled global illumi-

nation values need to be merged against a full-resolution rendered image. Usually it happens after the primary filtering stage, when the denoised downscaled probe values are stretched and interpolated over all pixels on the screen. Then, the blurred full-screen GI is integrated into the composite of full-resolution rendered buffers.

In the final step, the rendered image may require additional temporal filtering (TF), which hides the traces of jittered samples and provides a smooth and clear output image. [40, 33]

Practical Part

During my study of lighting algorithms, I also spent some time on experimenting with lighting settings inside the Unreal Engine editor. For that purpose, I have prepared 2 scenes representing indoor and outdoor environments. Figure(5.14)

Figure 5.14: Interior and Exterior Scenes.

In order to preserve some measure of interactivity in the final program, I have payed additional attention to redirecting user input into the engine's level controlling structures. The way the user can adjust some lighting settings in the built version of the program is by using GUI interface.

The interface, therefore, contains features provided by Post Process Volumes. These local structures carry the main settings for all lighting modes. By switching between them, I got a chance to compare the visual output of every ray tracing mode.

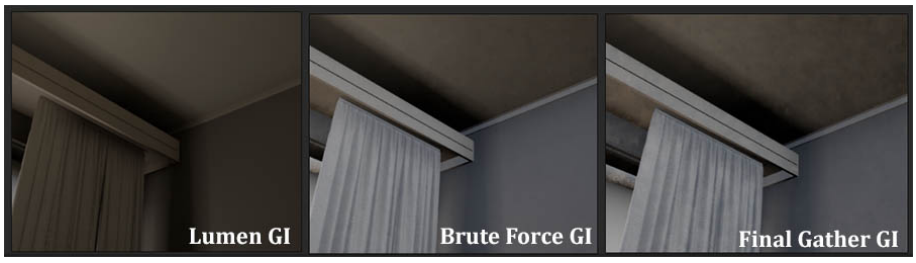

Figure 5.15: Global Illumination

As it was stated in the theoretical part, the impact of the global illumination was more noticeable in the indoor scene, in which the light was coming into the room through small windows.

Figure(5.15) shows screenshots of the room's corner lit by 3 global illumination (GI) methods.

the Lumen GI method, therefore, seemed to provide a very smooth output, that, nevertheless, had very low contrast and lacked deep shadows. It looked like all sampled values projected on the screen were averaged by large filtering kernel. This visual effect might indeed correspond to the spatial filtering performed on sparse screen gathering probes.

The hardware ray tracing solutions, on the other hand, produced a crisp and high contrast image enhanced by deep shadows. However, there was a substantial amount of noise and flickering, as the screen denoiser seemed to be failing at approximating light samples at real time rates. Yet, the main visual difference between the Brute Force and Final Gather approach was the amount of noise visible on the surfaces. The Final Gather method produced a lot worse output, but rendered a bit faster compared to the Brute Force approach.

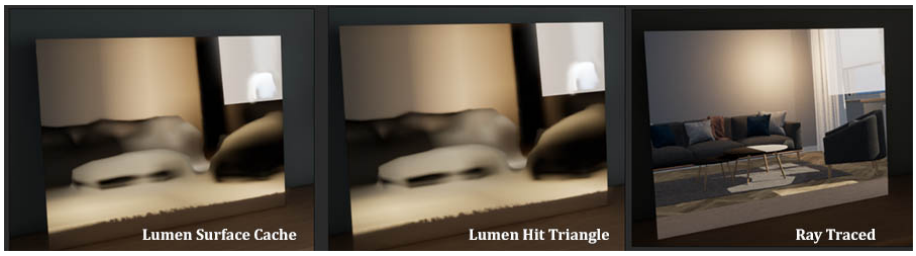

Figure 5.16: Reflections in the perfect mirror

Figure(5.16) illustrates the frames with rendered reflections. The first 2 were produced by Lumen Reflections system. I have tried to switch between 2 reflection modes; however, the output remained the same and showed the LumenScene in the reflection. The hardware ray tracing method however, returned, a very precise representation of the scene sampled by precise triangle hit tests.

Although, what I consider to be more important is that all lighting methods react differently to values of light emitters. I only had 3 light emitters in the scene represented by the Sunlight, the Skylight and

a single point light. However, every switch between both the GI and Reflection methods required tweaking parameters of every lighting source for the scene to look more natural.

By the end of my lighting test, I felt rather surprised by the small number of settings available for each lighting mode. It seemed like developers wanted to reduce the workload placed on the artists by making the lighting system self-driven. Yet, due the low number of available options, the user might find it difficult to control, while setting light scenarios required in the game production.

Conclusion

The latest advances in real-time rendering led to a significant increase of visual realism observed in modern games. In order to understand the scope and value of new lighting approaches presented in recent years, I firstly attempted to provide the measure of realism by comparing the lighting concept known from physics to the lighting model used in computer graphics. The comparison uncovered a series of compromises lying in the core of all the rendering methods, for which they are called PBR or physically-based methods.

The I proceeded by taking a deeper view on the ray tracing methods, as they are believed to be the future of high quality rendering. Nevertheless, following the ray tracing development history, I came to the conclusion that the ray tracing technique by itself does not guarantee realistic output. Confirmed by many implementations, the ray tracing methods are mainly chosen for their fast approximation of the light spread across the scene. Yet, the physical properties of the environment seem to be simulated by PBR shaders, which evaluate the strength of light's impact on the objects.

Since the PBR shaders are considered to be an established component in the rendering pipeline, the only way to further improve the visual output for dynamic scenes is to address the speed of light evaluation. By introducing the hardware support for ray tracing operations NVIDIA raised a further wave of interest among the game developers, persuading the members of the Epic's team to implement new DXR pipeline inside Unreal Engine. In my study of their attempts I have discovered, that 2 ray tracing methods, currently available in Unreal Engine, are capable of computing global illumination in real time. However, my practical experiments showed that both methods are severely limited by performance constraints, as their final output was quite noisy, and every quality improvement had a significant computational cost.

For that reason, I took a larger interest in Lumen, which is presented by Epic as a new dynamic lighting system for Unreal Engine 5. After my analysis of Lumen's pipeline I consider it to be another iteration on Unreal ray tracing techniques, which replaces the original hardware-accelerated functions by their software equivalents. The key

to Lumen's success, therefore, can be seen in the amount of control developers gained over the ray tracing pipeline, as they continue to optimize almost every calculation stage to decrease the performance costs.

So, the ray tracing approach the Epic's team chose to implement in the UE5 is very unique. However, I found it quite difficult to use. Epic did not expose many Lumen settings through the engine's interface, and the lack of Lumen associated view modes complicates debugging of unclear issues. Even console commands, which claim to control certain rendering stages, do not provide a visible feedback. However, there is a hope for Lumen to become more responsive and transparent in future engine's releases, as this lighting system is still in active development and new features tend to appear every year.

A An appendix

A simple application made in Unreal Engine 5.0.3 provides simple GUI controls to test the ray tracing methods described in the thesis. The program contains 2 small scenes, in which the user would be able to adjust the lighting settings to see their impact on the indoor and outdoor environments.

The application can be downloaded from the MUNI Thesis Archive.

Bibliography

1. GREGORY, Jason. *Game Engine Architecture*. 2nd ed. Cambridge (Mass): CRC Press, 2015. ISBN 978-1-4665-6006-2.
2. *List of common shading algorithms* [online]. Wikipedia [visited on 2022-11-17]. Available from: https://en.wikipedia.org/wiki/List_of_common_shading_algorithms.
3. VRIES, Joey de. *Basic lighting* [online]. 2014. [visited on 2022-11-12]. Available from: https://learnopengl.com/Lighting/Basic-Lighting.
4. *Light* [online]. Wikipedia [visited on 2022-11-10]. Available from: https://en.wikipedia.org/wiki/Light.
5. WHITTED, Turner. An Improved Illumination Model for Shaded Display [online]. 1980, vol. 23 [visited on 2022-11-04]. Available from: https://dl.acm.org/doi/10.1145/358876.358882.
6. *Photon Energy* [online]. Wikipedia [visited on 2022-11-13]. Available from: https://en.wikipedia.org/wiki/Photon_energy.
7. JEFF, Russel. *Basic Theory of Physically-Based Rendering* [online]. [visited on 2022-11-03]. Available from: https://marmoset.co/posts/basic-theory-of-physically-based-rendering/.
8. *The PBR Guide by Allegorithmic - Part 1: Light and Matter: The theory of Physically-Based Rendering and Shading* [online]. [visited on 2022-10-03]. Available from: https://substance3d.adobe.com/tutorials/courses/the-pbr-guide-part-1.
9. *The PBR Guide by Allegorithmic - Part 2: Practical Guidelines for Creating PBR Textures* [online]. [visited on 2022-10-03]. Available from: https://substance3d.adobe.com/tutorials/courses/the-pbr-guide-part-2.
10. JOE, Wilson. *Physically-Based Rendering: And You Can Too!* [online]. [visited on 2022-11-03]. Available from: https://marmoset.co/posts/physically-based-rendering-and-you-can-too/.
11. MOLLER, Tomas. *Real-Time Rendering*. 4th ed. CRC Press, 2018. ISBN 978-1-1386-2700-0.

12. VRIES, Joey de. *PBR: Theory* [online]. 2014. [visited on 2022-11-22]. Available from: https://learnopengl.com/PBR/Theory.

13. *Rendering equation* [online]. Wikipedia [visited on 2022-11-12]. Available from: https://en.wikipedia.org/wiki/Rendering_equation.

14. KAJIYA, James. The Rendering equation [online]. 1986, vol. 20 [visited on 2022-11-02]. ISBN 978-0-89791-196-2. Available from: https://dl.acm.org/doi/10.1145/15886.15902.

15. EVANSON, Nick. Path Tracing vs. Ray Tracing, Explained [online]. 2022 [visited on 2022-11-16]. Available from: https://www.techspot.com/article/2485-path-tracing-vs-ray-tracing/#:~:text=Traditional%5C%20ray%5C%20tracing%5C%20involves%5C%20calculating,off%5C%20in%5C%20a%5C%20random%5C%20direction.

16. ALAN WATT, Mark Watt. *Advanced Animation And Rendering Techniques: Theory and Practice*. Addison-Wesley, 1992. ISBN 0-201-54412-1.

17. COOK, Robert. Distributed Ray Tracing [online]. 1984, vol. 18 [visited on 2022-11-02]. Available from: https://dl.acm.org/doi/10.1145/964965.808590.

18. *Monte Carlo method* [online]. Wikipedia [visited on 2022-11-17]. Available from: https://en.wikipedia.org/wiki/Monte_Carlo_method.

19. *UE4 Documentation: Real-Time Ray Tracing* [online]. Epic [visited on 2022-11-19]. Available from: https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/RayTracing/.

20. *DirectX RayTracing (DXR) Functional Spec* [online]. [visited on 2022-10-28]. Available from: https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html#ray-geometry-interaction-diagram.

21. EDWARD LIU Llamas Ignacio, Kelly Patrick. Cinematic Rendering in UE4 with Real-Time Ray Tracing and Denoising. In: [online]. 2019 [visited on 2022-11-29]. Available from: https://link.springer.com/chapter/10.1007/978-1-4842-4427-2_19.

22. PATRCIK KELLY, Yuriy O'Donnell. Ray Tracing in Fortnite. In: [online]. 2021 [visited on 2022-11-16]. Available from: https://link.springer.com/chapter/10.1007/978-1-4842-7185-8_48.

23. CHRIS WYMAN, Marrs Adam. *Introduction to DirectX Raytracing* [online]. [visited on 2022-11-29]. Available from: https://www.researchgate.net/publication/354065088_Ray_Tracing_in_Fortnite.

24. *UE5 Documentation: Ray Tracing Performance Guide in Unreal Engine* [online]. Epic [visited on 2022-10-01]. Available from: https://docs.unrealengine.com/5.0/en-US/ray-tracing-performance-guide-in-unreal-engine/.

25. *UE5 Documentation: Lightmass Basics* [online]. Epic [visited on 2022-10-15]. Available from: https://docs.unrealengine.com/5.0/en-US/lightmass-basics-in-unreal-engine/.

26. *UE5 Documentation: Understanding Lightmapping in Unreal Engine* [online]. Epic [visited on 2022-10-16]. Available from: https://docs.unrealengine.com/5.0/en-US/understanding-lightmapping-in-unreal-engine/.

27. MCGUIRE, Morgan. *Ray-Traced Irradiance Fields* [online]. NVIDIA, 2019 [visited on 2022-11-13]. Available from: https://www.gdcvault.com/play/1026182/.

28. *UE5 Documentation: Indirect Lighting Cache* [online]. Epic [visited on 2022-10-15]. Available from: https://docs.unrealengine.com/5.0/en-US/indirect-lighting-cache-in-unreal-engine/.

29. *UE5 Documentation: Volumetric Lightmaps* [online]. Epic [visited on 2022-10-15]. Available from: https://docs.unrealengine.com/5.0/en-US/volumetric-lightmaps-in-unreal-engine/.

30. *UE5 Documentation: Path Tracer* [online]. Epic [visited on 2022-11-18]. Available from: https://docs.unrealengine.com/5.0/en-US/path-tracer-in-unreal-engine/.

31. *UE5 Documentation: Lumen Global Illumination and Reflections* [online]. Epic [visited on 2022-10-14]. Available from: https://docs.unrealengine.com/5.1/en-US/lumen-global-illumination-and-reflections-in-unreal-engine/.

32. *UE5 Documentation: Lumen Technical Details* [online]. Epic [visited on 2022-11-03]. Available from: https://docs.unrealengine.com/5.0/en-US/lumen-technical-details-in-unreal-engine/.

33. KELLY, Daniel Wright; Krzysztof Narkowicz; Patrick. *Lumen: Real-time Global Illumination in Unreal Engine 5* [online]. SIGGRAPH, 2022 [visited on 2022-10-28]. Available from: https://advances.realtimerendering.com/s2022/index.html.

34. STACHOWIAK, Tomasz. *Stochastic Screen-Space Reflections* [online]. Electronic Arts / Frostbite, 2015 [visited on 2022-11-23]. Available from: https://www.ea.com/frostbite/news/stochastic-screen-space-reflections.

35. *Hierarchical-Z map based occlusion culling* [online]. RasterGrid, 2010 [visited on 2022-11-22]. Available from: https://www.rastergrid.com/blog/2010/10/hierarchical-z-map-based-occlusion-culling/.

36. GARCIA, Maxime. *An Introduction to Raymarching* [online]. [visited on 2022-11-24]. Available from: https://typhomnt.github.io/teaching/ray_tracing/raymarching_intro/.

37. AREEJ. Unreal Engine 5 Lumen vs Ray Tracing: Which One is Better? [online]. 2022 [visited on 2022-10-03]. Available from: https://www.hardwaretimes.com/unreal-engine-5-lumen-vs-ray-tracing-which-one-is-better.

38. *UE5 Lumen Implementation Analysis* [online]. [visited on 2022-10-14]. Available from: https://blog.en.uwa4d.com/2022/01/25/ue5-lumen-implementation-analysis/.

39. *UE5 Documentation: Nanite* [online]. Epic [visited on 2022-10-16]. Available from: https://docs.unrealengine.com/5.0/en-US/nanite-virtualized-geometry-in-unreal-engine/.

40. WRIGHT, Daniel. *Radiance Caching for Real-Time Global Illumination* [online]. SIGGRAPH, 2021 [visited on 2022-11-02]. Available from: `https://advances.realtimerendering.com/s2021/index.html`.

41. BOISSE, Guillaume. GI-1.0: A Fast Scalable Two-Level Radiance Caching Scheme for Real-Time Global Illumination [online]. 2022 [visited on 2022-11-14]. Available from: `https://gpuopen.com/download/publications/GPUOpen2022_GI1_0.pdf`.