FREIE UNIVERSITÄT BERLIN

MASTER'S THESIS

Hardware Acceleration of Progressive Refinement Radiosity using Nvidia RTX

Author:
Benjamin KAHL

Supervisor:
Prof. Dr. Günter ROTE

Revised and corrected version of a thesis submitted in fulfillment of the requirements
for the degree of Master of Science
on

October 13, 2022

in the

Department of Mathematics and Computer Science

March 26, 2023

FREIE UNIVERSITÄT BERLIN

Abstract

Institute of Computer Science
Department of Mathematics and Computer Science

Master of Science

Hardware Acceleration of Progressive Refinement Radiosity using Nvidia RTX

by Benjamin KAHL

A vital component of photo-realistic image synthesis is the simulation of indirect diffuse reflections, which still remain a quintessential hurdle that modern rendering engines struggle to overcome. Real-time applications typically pre-generate diffuse lighting information offline using *radiosity* to avoid performing costly computations at run-time.

In this thesis we present a variant of *progressive refinement radiosity* that utilizes Nvidia's novel RTX technology to accelerate the process of form-factor computation without compromising on visual fidelity. Through a modern implementation built on DirectX 12 we demonstrate that offloading radiosity's visibility component to *RT cores* significantly improves the lightmap generation process and potentially propels it into the domain of real-time.

Contents

Chapter 1

Preface

The computational synthesis of photorealistic images has been a quintessential challenge in the computer graphics domain since its inception. Growing industries such as video games, virtual reality and visual effects have induced a veritable explosion in demand for increased realism over the last few decades. A major step towards this goal was taken by James Kajiya in 1989 when he formulated the *rendering equation* [13], which provides a general mathematical description of how light propagates through a 3D environment.

Unfortunately, the rendering equation proved far too complex to solve linearly. Every surface can receive light from infinitely many directions and then scatter it diffusely, effectively qualifying the surface as a separate light-source itself.

Computer graphics researchers have spent a large part of their endeavour grappling with the conundrum that is finding an ideal, numerically solvable model to this infinitely recursive complexity. Indeed, rendering algorithms we see employed today can all be regarded as approximations, shortcuts or simplifications of the rendering equation.

With regard to *global illumination*, two of these have stood the test of time: *ray tracing* for the generation of individual, highly realistic images and *radiosity* for real-time use cases that continuously render the same, static geometry from a large set of camera angles.

1.1   Nvidia RTX

Over the last decades raytracing has generally found its place as a crude and expensive approach that nevertheless provides a very high degree of photorealism, albeit at a proportionally high cost in required computation time.

Yet in 2018, fifty years after the first computer-based ray-tracer was created [14], the American tech company *Nvidia* unveiled their *GeForce RTX* series of graphics cards. Uniquely, these contain specialized computation units that can speed up raytracing-related operations to such a degree that it propels this blunt, brute-force approach into the domain of real-time [49].

The mathematical challenges faced by raytracing and radiosity are fundamentally identical and thus inextricably linked. In this thesis we argue that the considerable performance increase enabled by the RTX platform ought to be reflected in radiosity to the same degree it is seen in raytracing.

1.2 Motivation

Global illumination solutions based on radiosity typically generate lighting information and then export it into a texture, which can be rendered a-posteriori within consumer applications at virtually no cost at all. Despite great rendering performance, the process of generating these textures remains a computationally expensive process that can severely hamstring the development and design process of complex 3D environments.

Radiosity's performance bottleneck unequivocally lies with the vast amount of visibility calculations required [1]. Although this problem is, in theory, highly parallelizable [15], implementations of the radiosity model seem to generally favour multi-core CPUs (approx. 4-16 high performance cores) over GPUs (approx. 1-10 thousand low performance cores) [16], because GPU variants rely on hemicuboid z-Buffering for visibility determination [17].

In this thesis we investigate if an RTX-based visibility solution provides a performance improvement sufficient enough to fully advance radiosity into the realm of GPUs and parallel computing.

Not only is raytracing a highly adequate solution for visibility, but RTX GPUs also perform their operations on dedicated hardware in the form of a moderate amount (30 to 80) of highly specialized *RT cores* [49]. This intermediate solution between the parallelization levels of a CPU and a GPU may prove ideal for the acceleration of the radiosity algorithm.

1.3 Objective

The intended goal behind this thesis is to further the acceleration of radiosity computations for developers and designers of 3D environments working on machines compatible with RTX. Once lighting textures have been generated, they can in turn be rendered on almost any graphics hardware, regardless of RTX compatibility.

To accomplish this, we grapple a common variant of radiosity found on GPUs, known as *progressive refinement radiosity*, and substitute its z-Buffering components with an RTX-based approach. We will also investigate and examine potential performance improvements in addition to how well this approach compares to already existing solutions.

1.4 Thesis Structure

The next chapter will cover the theoretical knowledge required for the remainder of the thesis by deriving the rendering equation and providing mathematical models for several global illumination solutions.

Afterwards, chapter 3 takes a deep dive into RTX technology by examining and reviewing the underlying *Turing architecture* and *DirectX* raytracing pipeline.

Chapters 4 and 5 will present *RTRad*, our RTX-accelerated progressive refinement radiosity implementation as well as any related tweaks and potential performance improvements.

Lastly, in chapter 6, we compare and analyze the performance of this implementation upon which we draw our conclusions in chapter 7.

Chapter 2

Introduction

This chapter outlines the background knowledge and core concepts that are required in the subsequent chapters. The first section commences by deriving the core problem of computer graphics starting at the root. Afterwards, we show how the global illumination problem is tackled specifically by radiosity and raytracing.

2.1   The Speed - Realism Dichotomy

A classical image-synthetization process computes how light scattered into an environment translates into pixel colors on a retina. The color an object should adopt on a virtual sensor can be traced back to the wavelengths absorbed by its surface in relation to the light incident on it, which is turn affected by the light that is reflected, refracted or emitted by other surfaces around it.

Combine this endless recursion with the vast amount of intrinsics this process is subject to, such as physical properties or geometric arrangements, and it quickly becomes clear that a complete, physically accurate light simulation is an unfeasible computational task that needs to be approximated.

Indeed, even highly photorealistic, computationally heavy methods employ a significant amount of approximation and reductionism. The question therefore becomes which simplifications one is willing to make and what their payoff is in computational expense.

The balance of speed vs. realism that underpins this challenge divides it into two distinct problem domains: Whilst some industries, like CG film-making and SFX, are more geared towards realism, other areas have driven an increased demand of faster, more responsive graphics, known under the umbrella term of *real-time rendering*.

2.2   Rendering Optics

The essence of generating images from abstract descriptions can be narrowed down to the simulation of a real-world camera in a virtual environment. As such, we commence by examining the characteristics of virtual cameras as well as the related concepts from *radiometry* that help us model light propagation.

2.2.1   Camera Optics

Most genuine cameras have a series of common denominators arranged in a similar construction:

An aperture allows light to enter through a convex lens, which casts an image onto a light-capturing sensor. The convexity of the lens ensures that the direction in which light hits the sensor is restricted, thus focusing the image with a limited depth of field, determined by the focal length.

This directional limitation of incident light can also be accomplished without a lens, by severely limiting the size of the camera's aperture, which is the principle of the *pinhole camera* [74] (as seen in fig. 2.1).
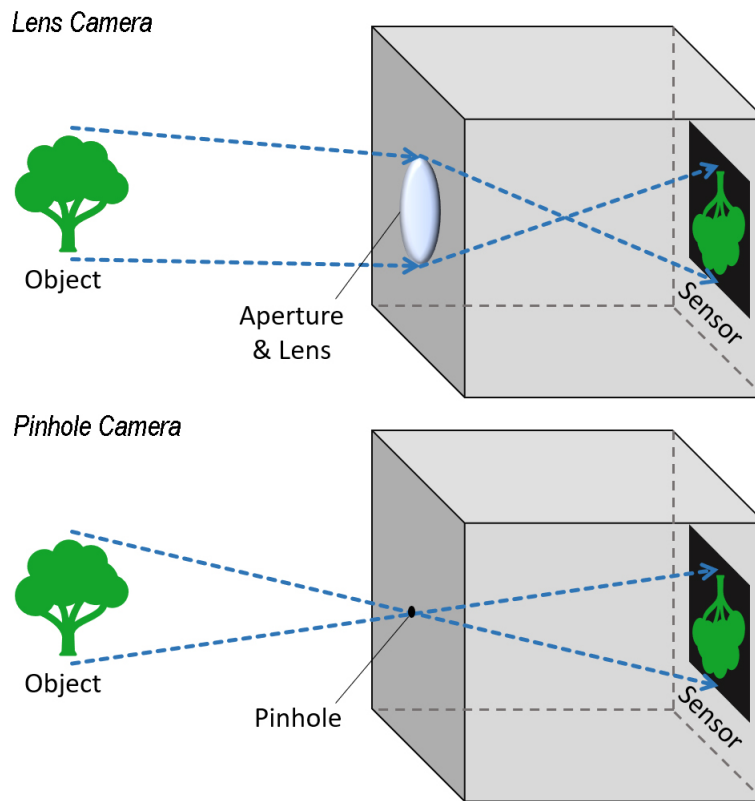
FIGURE 2.1: Comparison of the optics behind a lens-based and pinhole camera.

Pinhole cameras have nearly infinite depth of field and, unlike lens-based cameras, do not suffer from lens distortion (see fig. 2.2). However, their minuscule apertures require proportionally lengthy exposure times to produce serviceable photographs [74], for which reason pinhole cameras tend to find little to no use in real-life photography [18].

However, virtual environments are not subject to the same physical constraints, as any numerical value for light can arbitrarily be multiplied by some factor to control for brightness or exposure. As such, the notion of an *exposure time* is not a valid one within a virtual context.

2.2.2 Virtual Camera

A common observation one can make in computer-generated images is that they tend to have unlimited depth of field. This is, indeed, because virtual cameras strongly mimic the simplified optics of a pinhole camera [74].

In reality, increasing the distance between a pinhole and its sensor would produce a weaker image due to inverse-square attenuation. But since brightness factors and exposure times are irrelevant in a virtual context, the pinhole-sensor distance can be entirely discarded. As such, the sensor can be regarded as being a virtual screen *in front* of the camera [74], where light enters through grates corresponding to pixels on the final image. This arrangement is depicted in fig. 2.3.

FIGURE 2.2: Normal lens (left) vs. Pinhole lens (right). The pinhole
has greater depth of field, but the image sharpness decreases with
pinhole size. Image credited to Leonard Lessin/Science Source [60].

By designating the location of the pinhole as the camera's position, we are left
with a location vector, a view direction and two *field-of-view* (FOV) angles that are
proportional to the height and width of the resulting image respectively [1].

$$Camera = \{C, \vec{x}, \vec{y}, \vec{z}, \measuredangle_x, \measuredangle_y\} \tag{2.1}$$

where C is the location of the camera, $\vec{x}$, $\vec{y}$ and $\vec{z}$ are the camera's right, upward
and forward directions respectively and $\measuredangle_x$, $\measuredangle_y$ are the FOV angles for the x and y
directions.

In most practical cases, the given directions form an orthogonal coordinate system with $\hat{z} = \hat{x} \times \hat{y}$.
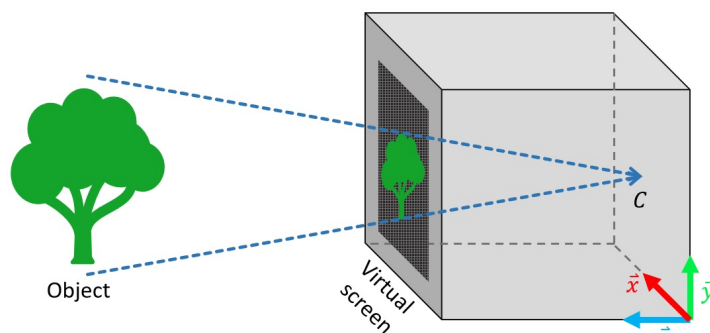
FIGURE 2.3: Optics of a virtual camera. The dimensions of the virtual
screen are directly tied to the angles $\measuredangle_x$, $\measuredangle_y$ making them independent
of their distance from C.

---

[1]Most computer graphics domains expand this definition by also including a near and far *clipping
plane*, thus forming a *view frustum* [74].

2.3 Radiometric Quantities

The process of rendering a virtual object from the perspective of a virtual camera now comes down to measuring the light that the object emits or reflects towards the location of the virtual pinhole C.

The direction of incident photons from a given surface cannot be described by a simple 3D vector, as surfaces subtend an infinite amount of directions towards a single point. Thus, it is useful to define a measurement of *solid angle* for the total *field of view* an object occupies towards an observer.

2.3.1 Solid Angle

Regular 2D angles can be adequately represented by the length of the arc they cover on a unit circle. Analogously, solid angles are proportional to the area a surface projects onto a unit sphere around the point of origin [1, 67]. Solid angles are measured in *steradians* and limited by the total surface area 4π of a unit sphere.

Let dA be a differential, arbitrarily rotated surface area at point x' with the normal vector $\vec{n}$. The solid angle that dA occupies at another point x can be calculated using two separate operations [1], both of which are illustrated in fig. 2.4.
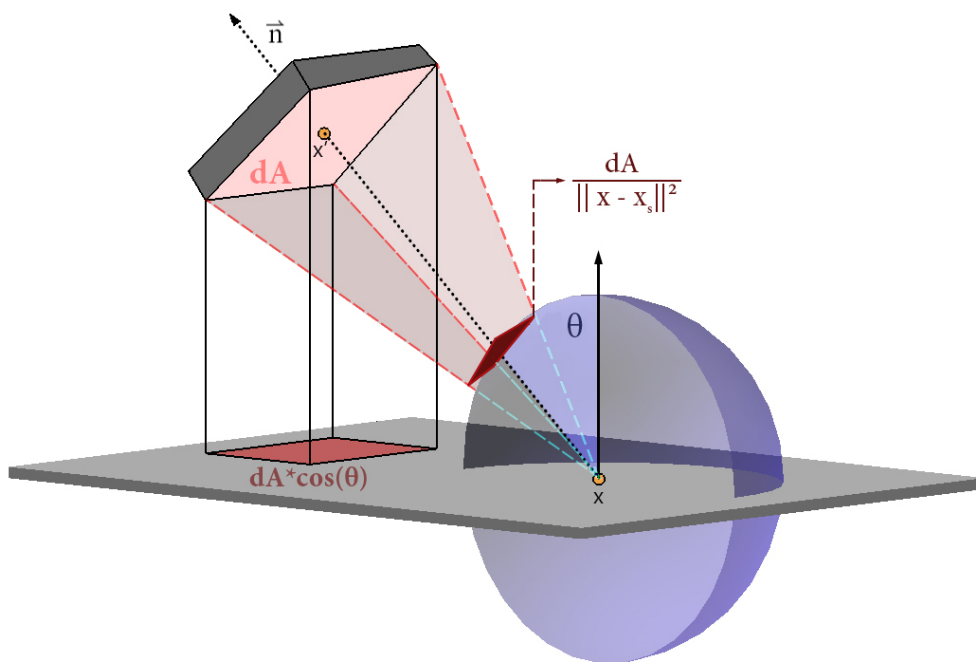
FIGURE 2.4: A surface projected onto a plane results in a surface area of $dA\cos\theta$. Projecting a perpendicular surface towards the center of a sphere results in a projected surface area on the sphere of $dA/\|x - x'\|^2$. Image from work by Benjamin Kahl [8].

- The surface area dA projects onto a plane perpendicular to $x' - x$ is equal to $dA\cos(\theta)$, where θ is the angle between $\vec{n}$ and $x' - x$ [1]. This operation accounts for the *rotation* of dA and gives us the surface area of dA that is perpendicular to x.

- Projecting a perpendicular, differential surface onto a unit sphere around x is simply given by the inverse square of their distance [1]. This operation is given by the *inverse square law* and accounts for the distance between x and x'.

Combining both these operations into a single formula, will take both rotation and distance into consideration [1, 68]:

$$\omega = \frac{\cos(\theta)dA}{\|x' - x\|^2} \tag{2.2}$$

The field-of-view that an object occupies on the sensor of a virtual camera can be quantified through a solid angle. What color the corresponding pixels should adopt now depends on the light that the object emits, reflects or refracts towards the camera, which is given by its *radiance*.

2.3.2 Radiance

Radiance describes the radiant energy propagating in a given direction $\vec{\omega}$ at a given location x.

Let $p(x, \vec{\omega}, \lambda)$ be the volume density of photons of wavelength λ at position x that are travelling in direction $\vec{\omega}$, then the corresponding radiance $L(x, \vec{\omega})$ equates to the product of said photon density and the energy of a single photon $\frac{hc}{\lambda}$, integrated over all wavelengths [1]:

$$L(x, \vec{\omega}) = \int_{\lambda} p(x, \vec{\omega}, \lambda) \frac{hc}{\lambda} \tag{2.3}$$

Since the retinas in human eyes consist of three different types of photoreceptor cones (red, green and blue respectively), pixel colors are usually modelled as 3D vectors with each dimension corresponding to a respective color component. A commonly employed format is the RGB 24-bit color depth format, which assigns each component 8 bits of depth, with another optional 8 bits for transparency in an *alpha channel*.

The overall magnitude of a color-vector is a measure of its overall energy, which corresponds to the radiometric quantity of *flux*.

2.3.3 Radiant Flux

In computer graphics the quantum nature of light (photon density) tends to be discarded in favour of *radiant flux* Φ, which amounts to a general measure of radiant energy per unit time [1, 68]:

$$\Phi = \frac{\partial Q}{\partial t} [W] \tag{2.4}$$

where Q is the energy emitted, transmitted or reflected.

The total flux a surface A emanates is equal to the total radiance all the points on this surface emit in all directions [1]:

$$\Phi_o = \int_{x \in A} \int_{\omega} L(x, \vec{\omega}) \tag{2.5}$$

Respectively, the radiance exiting a surface in a particular direction is the total flux the surface emits per unit solid angle per unit projected surface area [8]:

$$L = \frac{d\Phi}{d\omega dA_\perp} = \frac{d\Phi}{d\omega dA \cos\theta} \qquad (2.6)$$

This formulation of radiance is of particular importance, as it acts as a measure of how bright a surface would appear to a camera in direction $\vec{\omega}$ [1].

Henceforth we will refer to radiance *exiting* a surface point in a certain direction as $L_o(x, \vec{\omega})$ and radiance *incident* on that point from $\vec{\omega}$ as $L_i(x, \vec{\omega})$.

Under the assumption that a surface does not emit any light of its own, we can discern that the radiance L_o emitted towards a camera would have to be less or equal to the flux incident on the surface, as per conservation of energy. The total flux incident on a given surface can be quantified by the value of *irradiance*.

2.3.4 Irradiance and Radiant Exitance

The total flux incident on a surface per unit surface area is termed the *irradiance E* of that surface (sometimes known as *illuminance*) [1, 68]:

$$E = \frac{d\Phi_i}{dA} [W/m^2] \qquad (2.7)$$

It is equivalent to the radiance incident from all directions in a hemisphere Ω above the surface [1, 68]:

$$E = \frac{d\Phi_i}{dA} = \int_\Omega L_i(\omega) \cos\theta d\omega \qquad (2.8)$$

(a) Radiant Flux Φ

(b) Radiance L
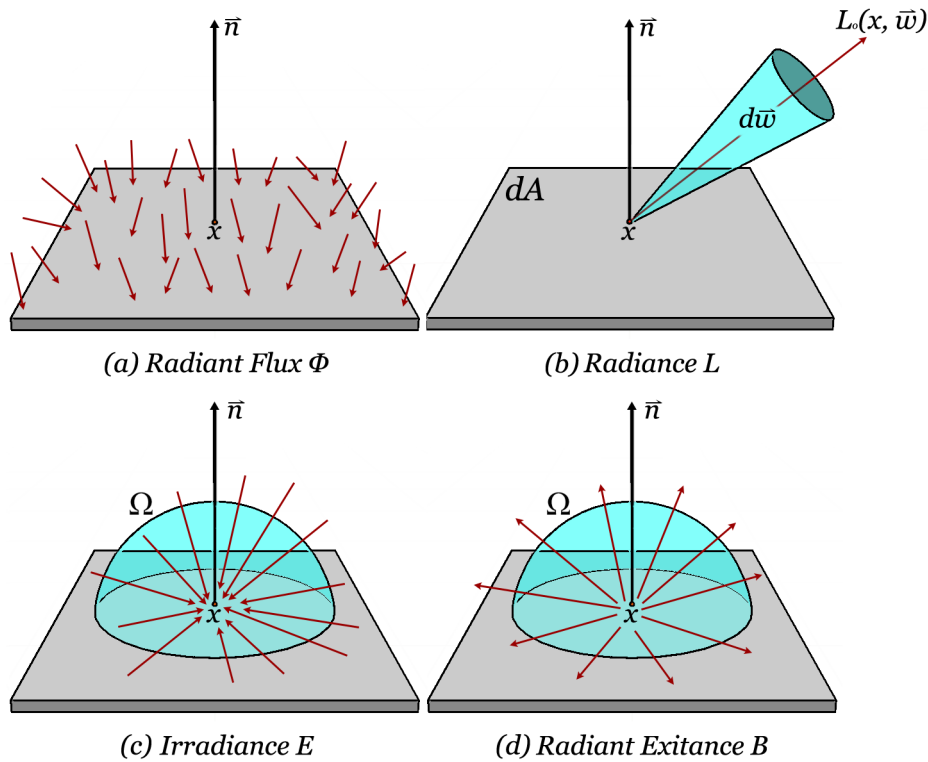
(c) Irradiance E

(d) Radiant Exitance B

FIGURE 2.5: Visualization of flux, radiance, irradiance and radiant exitance (a-d). Images based on depictions by Jarosz et al. [19].

The opposite of irradiance is the *radiant exitance B*, which is defined as the flux per surface area leaving or being emanated from a surface [2]:

$$B = \frac{d\Phi_o}{dA} = \int_\Omega L_o(\omega) \cos\theta \, d\omega \qquad (2.9)$$

The first law of thermodynamics dictates that energy is neither created nor destroyed. In the context of computer graphics it implies that the radiance reflected by a non-emissive surface must be less or equal to the radiance it receives.

Put differently, the radiance of a point x must be proportional to its irradiance under a coefficient of one or less [1]:

$$dL_o(\vec{\omega_o}) \propto dE(\vec{\omega_i}) \qquad (2.10)$$

The coefficient of proportionality that ordains this relationship is given by the *bidirectional reflectance distribution function* of x.

2.3.5 BRDF and the Reflectance Equation

For any given pair of differential solid angles (i.e. directions) $\vec{\omega_i}$ and $\vec{\omega_o}$, a material's *bidirectional reflectance distribution function* (BRDF) defines the ratio of flux concentration per steradian incident from $\vec{\omega_i}$ that is reflected into $\vec{\omega_o}$ [1, 67]:

$$f_r(\vec{\omega_i} \to \vec{\omega_o}) = \frac{L_o(\vec{\omega_o})}{E(\vec{\omega_i})} = \frac{L_o(\vec{\omega_o})}{L_i(\vec{\omega_i}) \cos\theta \, d\omega_i} \qquad (2.11)$$

If we solve this equation for L_o and perform the same hemispherical integral over the set of all incident directions Ω as in (2.8), we arrive at the total amount of light reflected by a surface in a specified direction, also known as the *reflectance equation* [1]:

$$f_r(\vec{\omega_i} \to \vec{\omega_o}) = \frac{L_o(\vec{\omega_o})}{E(\vec{\omega_i})}$$
$$\Leftrightarrow L_o(\vec{\omega_o}) = \int_\Omega f_r(\vec{\omega_i} \to \vec{\omega_o}) L_i(\vec{\omega_i}) \cos\theta_i \, d\omega_i \qquad (2.12)$$

Put simply, the reflectance equation prescribes that the radiance L_o a surface reflects in a particular direction $\vec{\omega_o}$ equates to its BRDF weighted irradiance.

2.4 The Rendering Equation

The reflectance equation yields the light a surface point reflects towards a camera. Including a term $L_e(x, \vec{\omega})$ for emission (light the point x emits itself) provides the total radiance the surface emanates in a direction $\vec{\omega}$ [1] (see fig. 2.6). This sum constitutes the *Rendering Equation*, as originally formulated by Kayija et al. in 1986 [13, 1]:

$$L_o(x, \vec{\omega}) = L_e(x, \vec{\omega}) + \int_\Omega f_r(\vec{\omega_i}, \vec{\omega}, x) L_i(x, \vec{\omega_i})(\vec{\omega_i} \cdot \vec{n_x}) d\omega_i \qquad (2.13)$$

The rendering equation states that "the transport intensity of light from one surface point to another is simply the sum of emitted light and the total light intensity which is scattered toward x form all other surface points" [13].
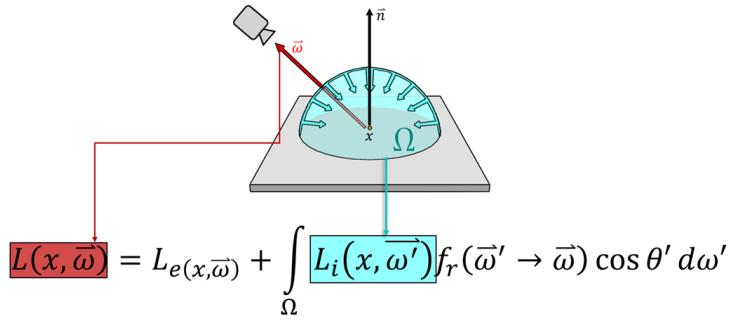
$$L(x, \vec{\omega}) = L_{e(x,\vec{\omega})} + \int_{\Omega} L_i(x, \vec{\omega'}) f_r(\vec{\omega'} \to \vec{\omega}) \cos \theta' \, d\omega'$$

FIGURE 2.6: The fundamentals of the rendering equation: *L* is a weighted integral over all incident directions.

An alternative formulation of this equation can be derived by replacing the hemispheric integral of directions by an integral over all other surface points [13, 1].

Let *S* be the set of all surfaces in the scene, then the irradiance *E* incident on a given surface point *x* is the integral over all light leaving any other surface point towards *x*, so long as *x* and the other point *x'* are mutually visible. The term of visibility - or occlusion - is given by a function $V(x, x')$ which is equal to 1 if *x* and *x'* are mutually visible, 0 if not:

$$E = \int_S L_o(x', \vec{\omega'}) V(x, x') \frac{\cos \theta \cos \theta'}{\|x - x'\|^2} dA \tag{2.14}$$

The multiplication of the two cosines $\cos \theta$ and $\cos \theta'$ accounts for the mutually projected surface area of the two locations and the division by the square of their distance stems from the inverse-square law.

Applying this equation for irradiance to the reflectance equation in (2.12) and adding the emissive component L_e, leaves us with the following variant of the rendering equation [13, 1]:

$$L_o(x, \vec{\omega}) = L_e(x, \vec{\omega}) + \int_S f_r(\vec{\omega'}, \vec{\omega}, x) L_o(x', \vec{\omega'}) V(x, x') \frac{\cos \theta \cos \theta'}{\|x - x'\|^2} dA \tag{2.15}$$

For the sake of simplicity, we will henceforth refer to these individual variants as the *hemispheric-* and *surface-based* rendering equation respectively.

2.5 Specular and Diffuse BRDFs

Materials we encounter in reality tend to be highly granular and possess immense detail on a microscopic scale, which leads to light being reflected in complex distributions of outgoing directions that our simple, computational models cannot fully replicate.

The tendency in computer graphics is to differentiate between three distinct reflectance components: diffuse, specular and glossy [1, 8], as portrayed in fig. 2.7 and fig. 2.8. These components are weighed in various proportions to one another, depending on the underlying material's physical properties and parameters.
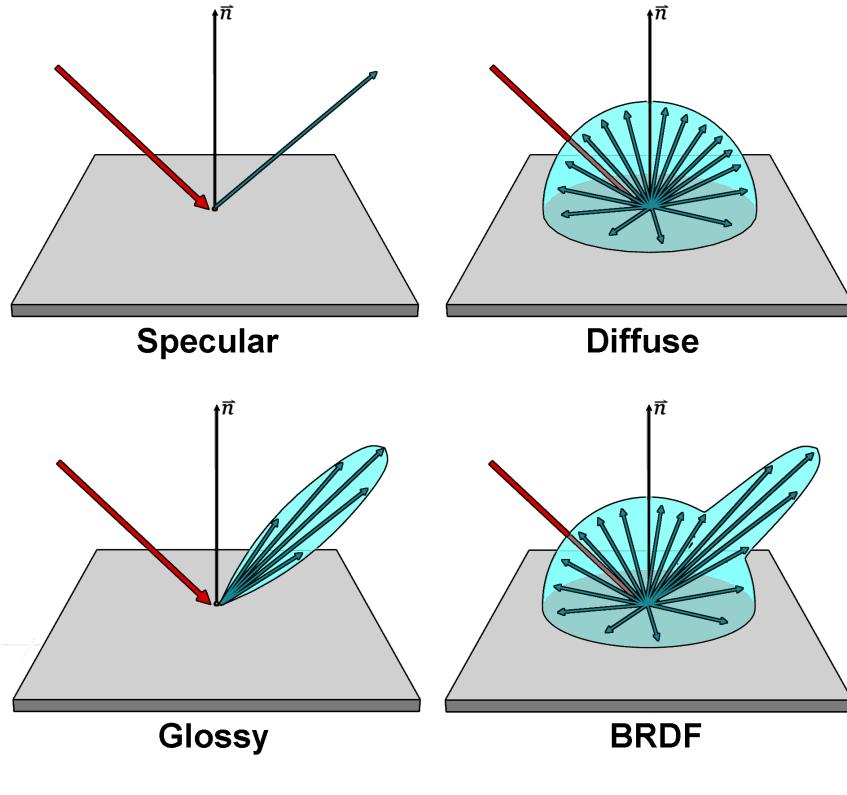
FIGURE 2.7: Reflectance components as part of a BRDF for a given
angle of incidence (in red).

2.5.1 Specular Reflections

In a *specular* reflection the incident lights' trajectory is perfectly mirrored across the surface's normal vector. We can model this behaviour by applying a *dirac delta function* to the angle around the normal [1]:

$$f_r(\vec{\omega}_i \to \vec{\omega}_o) = \frac{\delta(\cos\theta_i - \cos\theta_o)}{\cos\theta_i}\delta(\phi_i - (\phi_o \pm \pi)) \tag{2.16}$$

The values of θ and ϕ correspond to the angles with and around the upward axis respectively. The delta function δ yields zero for any non-zero parameter.

2.5.2 Glossy Reflections

Glossy reflections typically describe specular reflections with a small to moderate amount of scattering and variation. A material's *glossiness* determines the degree of diffusion that occurs.

Mathematically, a broader version of the delta function can be modelled by taking the dot product of two normalized vectors and raising it to some high exponent [8, 75].

Let $\vec{R}$ be the vector of a perfectly specular reflection and $\vec{\omega}_o$ be a vector that points from a surface towards the camera, then the glossy radiance equates to the following [8, 75, 20]:

$$L_o(\vec{\omega}_o) = L_i d\omega_i(\max(\vec{R} \cdot \vec{\omega}_o), 0)^\alpha \tag{2.17}$$

The value of $\max(\vec{R} \cdot \vec{\omega}_o), 0)$ is zero for angles larger than 90° and increases if $\vec{R}$ and $\vec{\omega}_o$ are of similar angle to the surface normal. The glossiness α dictates the narrowness of the underlying pseudo-delta function.

diffuse + *glossy* + *specular* =

FIGURE 2.8: Reflectance components on a sphere rendered in the Blender 2.8 EEVEE engine.

2.5.3 Diffuse Reflections

As depicted in fig. 2.7, *labertian diffuse* reflections are defined as completely isotropic, where the apparent brightness and surface color remain equal for all angles of observation [1]:

$$\forall \alpha, \beta \in \Omega : \quad f_r(x \to \alpha) = f_r(x \to \beta) \tag{2.18}$$

The corresponding BRDF f_r thus acts as a constant, since it retains the same value for all parameters.

Replacing the BRDF in the reflectance equation (2.12) by a constant allows for it to be separated from the integrand, providing us with a constant fraction of the irradiance E [1]:

$$\begin{aligned} L_o(\vec{\omega}_o) &= \int_{\Omega} f_r L_i(\vec{\omega}_i) \cos \theta_i d\omega_i \\ &= f_r \int_{\Omega} L_i(\vec{\omega}_i) \cos \theta_i d\omega_i \\ &= f_r E \end{aligned} \tag{2.19}$$

Intuitively, this equation describes that if the light incident on a surface gets scattered evenly, then the same fraction of the irradiance is reflected in all directions.

Since genuine surfaces typically absorb a portion of the light incident on them, it is useful to define a measure of *reflectivity* ρ that defines what percentage of irradiance is reflected into radiant exitance [1]:

$$\rho = \frac{B}{E} = \frac{\int_{\Omega_o} L_o(\vec{\omega}_o) \cos \theta_o d\omega_o}{E} \tag{2.20}$$

L_o is constant for all directions and Ω constitutes a hemisphere, as such, ρ constitutes the BRDF constant multiplied by π [1]:

$$\begin{aligned}
\rho &= \frac{\int_{\Omega_o} L_o(\vec{\omega}_o) \cos\theta_o d\omega_o}{E} \\
&= \frac{L_o \int_{\Omega_o} \cos\theta_o d\omega_o}{E} \\
&= \frac{L_o \pi}{E} \\
&= \pi f_r
\end{aligned}$$
(2.21)

Which, in turn, implies that the BRDF for a lambertian diffuse reflection is equal to a multiplication by $\frac{\rho}{\pi}$ [1]:

$$f_r = \frac{\rho}{\pi}$$
(2.22)

Diffuse reflections are the most difficult type of reflections to model accurately, as they require knowing the total light incident for *all* directions.

2.6 Rasterization

The rendering equation provides a mathematical model of what brightness and color a given surface adopts for a given observer. To compile this information into an actual image, we require the set of *pixels* on the image that a surface occupies.

In practice, three-dimensional scenes are usually described by a series of simple polygons - known as *primitives* - that span areas between 3D points known as *vertices*. The conversion process of a primitive into a corresponding set of pixels that it occupies on a screen is known as *rasterization* and forms a vital step in the majority of rendering pipelines. A more detailed account of this process is given in section 3.3.1.

The most common rasterization methods triangulate higher degree polygons and proceed to only rasterize pixels if their center lies completely inside a triangle. *Conservative rasterization* can add some certainty to pixel rendering, as all pixels that are at least partially covered by a rendered primitive are rasterized [50] (see fig. 2.9).
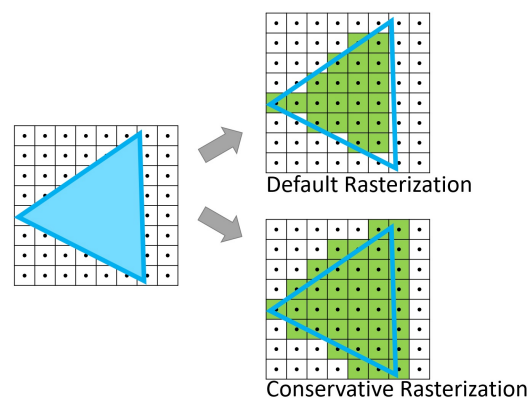

FIGURE 2.9: Default and conservative rasterization in comparison: Default rasterization only rasterizes fragments if their center is covered by the triangle. Conservative rasterization additionally includes all fragments partially covered by the triangle.

Once the pixels a surface occupies have been determined, a plethora of different lighting models can be applied to compute the color for each pixel individually, such as the *Phong local illumination model* [20]. So called *global illumination* models will include indirect light, i.e. light that bounces more than once before reaching the camera (see fig. 2.10).

All illumination models are, in essence, an exercise in solving the rendering equation through approximation.
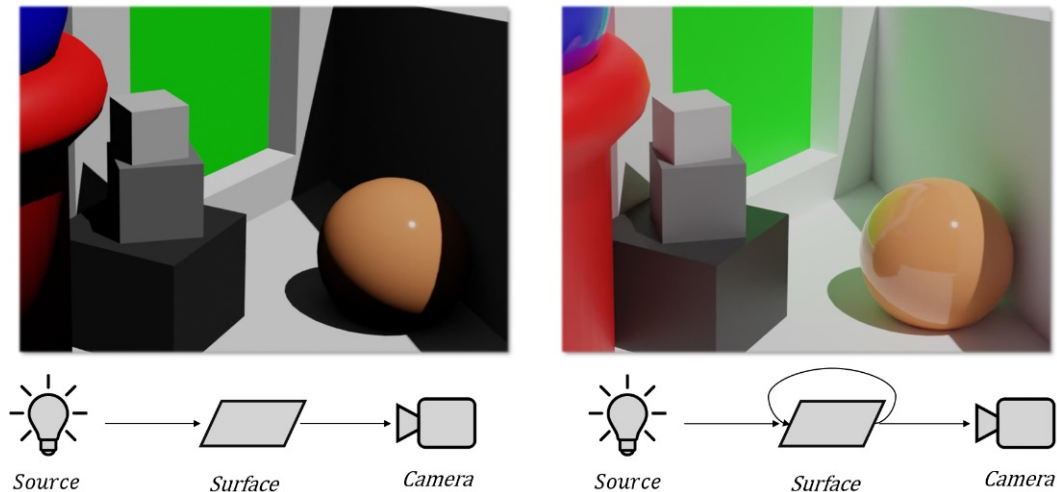
FIGURE 2.10: Local illumination (left) vs. global illumination (right) in the Unity 2020.2 Engine. Note in particular the indirect, green light on the sphere and angled surfaces.

2.7 Raytracing

The infinite recursion and integration of the rendering equation might appear to have one simple solution: Apply a maximum recursion-depth limit and approximate the hemispheric integral as a finite sum of directions. This exact thought-process lies behind *raytracing*, which has long stood as one the simplest and most intuitive approaches of numerically solving the rendering equation.

The core concept is simple and consists of tracing rays starting from the camera position through each pixel of the virtual screen and then setting the pixel colors based on the surfaces the respective rays collide with [67, 3]. Shooting rays directly from the camera makes a discrete rasterization unnecessary.

Computing a surface's lit color usually involves launching further sets of rays, which depend on the surface's material properties as well as the specific raytracing variant that is being utilized [3]. For instance, if the intersection surface is smooth, a specular ray is cast in the reflection direction and whichever surface this *specular ray* collides with will be mirror-reflected by the former surface. Rough surfaces, on the other hand, require sampling a myriad of rays to account for diffuse light (see fig. 2.11).

This process occurs recursively, with each intersection shooting its own set of rays until either a light source is hit or a maximum amount of bounces is reached. Whereupon the tree of rays is evaluated bottom-up until, at the root, the pixel's final color can be calculated [3].

FIGURE 2.11: Illustration of the raytracing process with the resulting image. Specular rays are marked in red, diffuse rays in green and initial rays in blue.

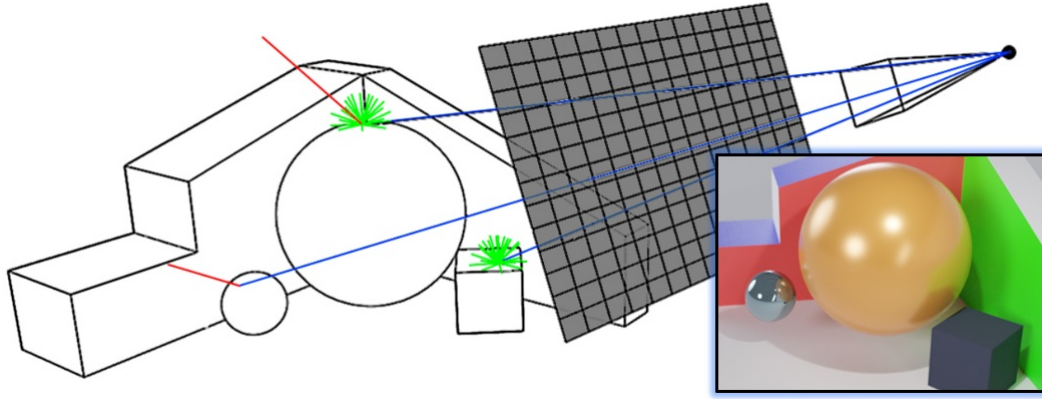This process is, in essence, a brute-force approach at computing the Riemann sum of the hemispheric rendering equation, as defined in (2.13):

$$L_o(x, \vec{\omega}) = L_e(x, \vec{\omega}) + \int_\Omega f_r(\vec{\omega_i}, \vec{\omega}, x) L_i(\vec{\omega_i}, x)(\vec{\omega_i} \cdot \vec{n_x}) d\omega_i \qquad (2.23)$$

Let $I(x, \vec{\omega})$ be a function that yields the closest surface intersected by a ray along $\vec{\omega}$ starting from x. The radiance incident at a surface point x from a given direction $\vec{\omega}$ must be equal to the light exiting the closest surface visible from $\vec{\omega}$ in the reverse direction taken to the inverse square of their distance:

$$L_i(\vec{\omega}, x) = \frac{L_o(I(x, \vec{\omega}), -\vec{\omega})}{\|x - I(x, \vec{\omega})\|^2} \qquad (2.24)$$

Based on this concept, raytracing collapses the hemispherical integral of the rendering equation into a finite sum of directions that are randomly and isotropically selected from the hemisphere:

$$L_o(x, \vec{\omega}) = L_e(x, \vec{\omega}) + \frac{1}{|\Omega|} \sum_{\vec{\omega_i} \in \Omega} f_r(\vec{\omega_i}, \vec{\omega}, x) L_o(I(x, \vec{\omega_i}), -\vec{\omega}) \frac{\vec{\omega_i} \cdot \vec{n_x}}{\|x - I(x, \vec{\omega_i})\|^2} \qquad (2.25)$$

where Ω is such a finite set of directions.

Despite being fundamentally simple, raytracing can produce highly photorealistic images because physical effects like refractions or caustics can easily be replicated. However, it does come at an equally high computational cost, as the recursive process can create a colossal amount of rays, all of which have to undergo a vast amount of intersection tests with the scene's geometry.

2.7.1 Ray Definition

Rays follow the mathematical description of a three-dimensional half-line constituted by an origin o and a direction $\vec{d}$ [3]. In its parametric form, a ray can be described as:

$$R(t) = o + t\,\vec{d} \quad \text{for } 0 \leq t < \infty \tag{2.26}$$

In practice it can be useful to compute cosines between vectors via dot-products, as is done in glossy reflections (see section 2.5.2). To facilitate this process, direction vectors are often restricted to be normalized unit vectors $\hat{d}$ [3].

This restriction also implies that the distance from the origin is directly represented by t. More generally, the difference in t value is equal to the distance between the respective points [3]:

$$\|R(t_1) - R(t_2)\| = |t_1 - t_2| \tag{2.27}$$
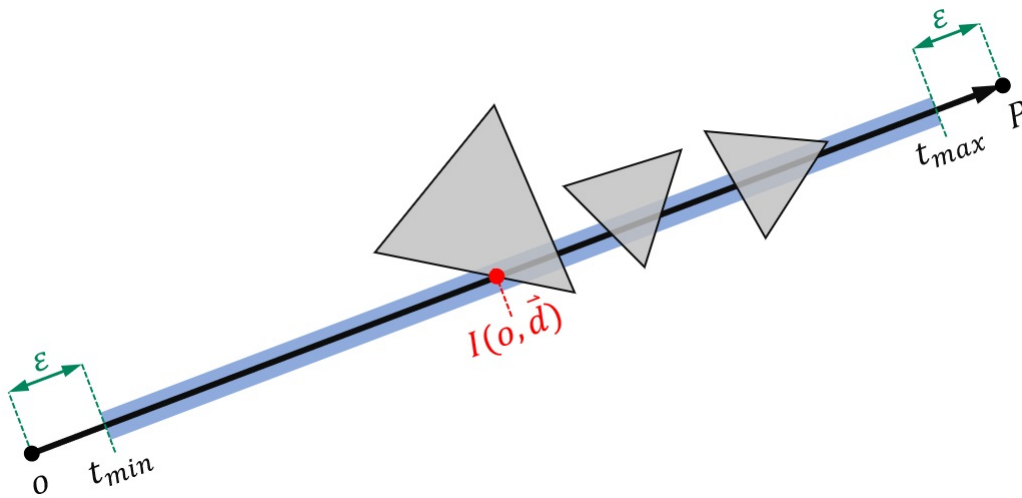
FIGURE 2.12: A ray launched from o to P. To avoid precision problems the interval is offset by ϵ. Of the three intersections the first is reported as the *closest hit*. Image based on a depiction from *Raytracing Gems* [3].

In raytracing, rays are intersected with a finite amount of geometry to determine collision points. As such, the semi-infinite description of a ray is not practical. Instead, rays frequently are defined with an additional interval that constitutes the range of t-values for which an intersection is useful: $t \in [t_{min}, t_{max}]$. Intersections that lie outside this interval are not reported (see fig. 2.12).

The utility of this interval comes in the form of several advantages: Firstly, the minimum value can help prevent self-intersections with the geometry itself that can arise from floating-point inaccuracies [3]. Secondly, the maximum value can speed up intersection calculations when hits beyond a certain point do not matter, like with a shadow ray that is shot towards a light-source [3].

2.7.2 Ray-triangle Intersection

Ray-triangle intersection tests are at the heart of the raytracing algorithm. Given a triangle T with the three vertices V_1, V_2 and V_3 as well as a ray R with origin o and direction $\vec{d}$, we seek the intersection point where R is equal to T.

One of the most commonly employed intersection algorithms in computer graphics is the *Möller-Trumbore* algorithm [21], which functions with minimal memory requirements by defining triangles in their parametric form of their *barycentric coordinates*:

$$T(u,v) = (1 - u - v)V_1 + uV_2 + vV_3 \tag{2.28}$$

where the barycentric coordinates u and v must fulfill the conditions $u \geq 0, v \geq 0$ and $u + v \leq 1$.
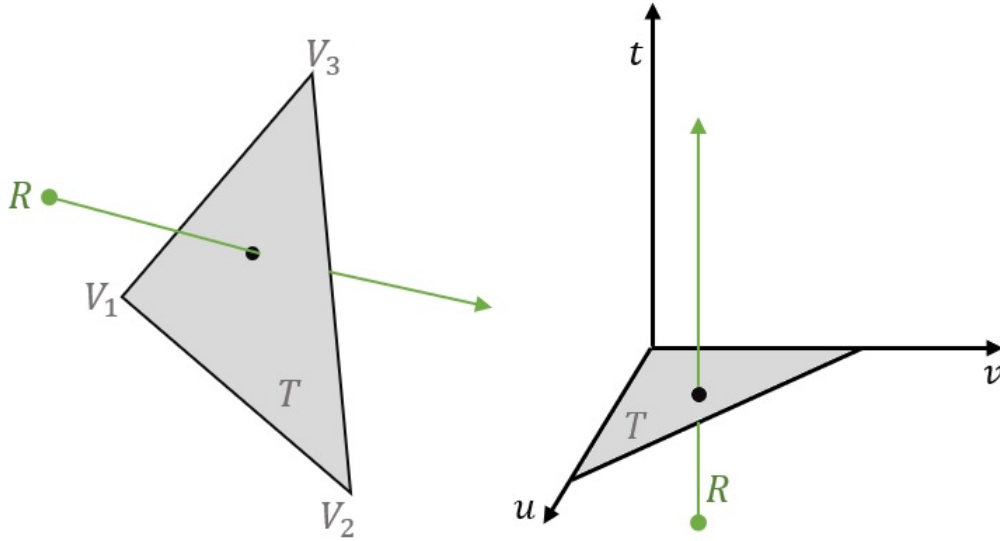
FIGURE 2.13: The system of linear equations produced in Möller-Trumbore algorithm effectively expresses the point of intersection in t, u, v space. The unit triangle spanned by u and v corresponds to T.
Image based on depiction by Scratchapixel [76].

Equating T with the definition for a ray put forth in (2.26) yields the following expression [21]:

$$\begin{aligned} R(t) &= T(u,v) \\ \iff o + t\vec{d} &= (1 - u - v)V_1 + uV_2 + vV_3 \\ \iff o + t\vec{d} &= V_1 + u(V_2 - V_1) + v(V_3 - V_1) \\ \iff o - V_1 &= -t\vec{d} + u(V_2 - V_1) + v(V_3 - V_1) \end{aligned} \tag{2.29}$$

which can be written as a linear system of equations [21]:

$$o - V_1 = \begin{bmatrix} -D & (V_2 - V_1) & (V_3 - V_1) \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} \tag{2.30}$$

This process is equivalent with expressing the point of intersection in a coordinate system spanned by the t, u and v axes, as depicted in fig. 2.13.

There are several methods for solving linear equation systems, but the one employed by Möller and Trumbore is *Cramer's Rule* [21].

The resulting t, u and v values give both the distance from the ray origin as well as the barycentric triangle coordinates of the intersection. If the conditions from

([2.28](#)) are not met or t lies outside of its defined interval, then no intersection exists [21].

Running this algorithm for every triangle in a scene whilst minimizing for t gives us the first - or closest - point that a ray encounters, which corresponds to the function I defined above:

$$I(o, \vec{d}) = o + t\,\vec{d} \quad \text{for } t \text{ being the minimal value in } [t_{min}, t_{max}] \text{ that satisfies}$$

$$R(o, \vec{d}) = T_i(u, v) \text{ with}$$
$$u \geq 0, v \geq 0 \text{ and } u + v \leq 1 \tag{2.31}$$
$$\text{for any triangle } T_i \text{ in the scene}$$
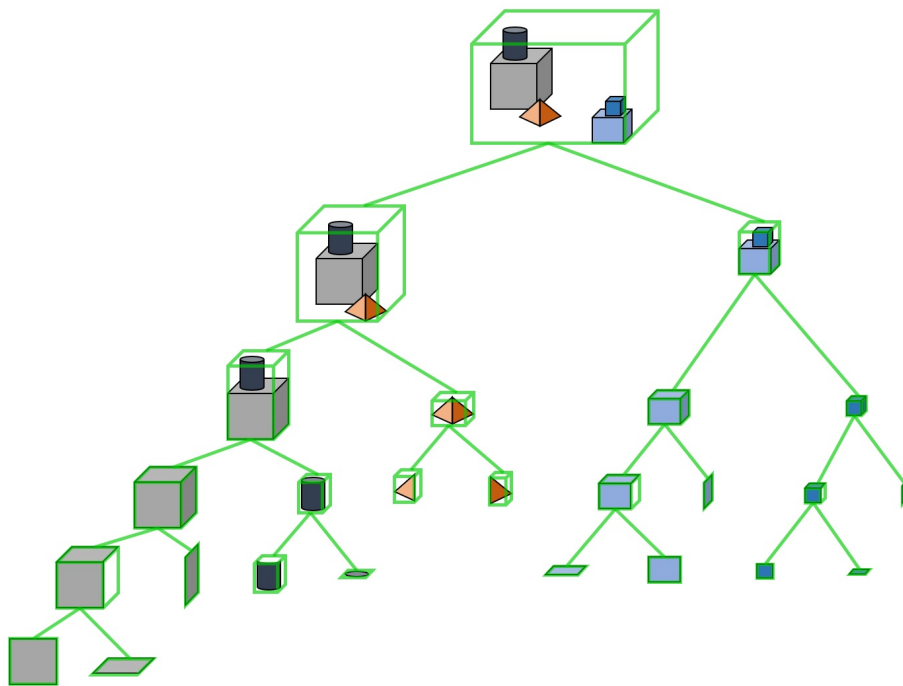
FIGURE 2.14: Simplified illustration of a BVH of a simple 3D scene.

2.7.3 Bounding Volume Hierarchies

The intersection tests required for tracing a ray can constitute a huge computational endeavour that *acceleration data structures* aim to ameliorate.

Primitives can be grouped into bounding boxes that can in turn be grouped as well, thereby forming a hierarchy [3], as depicted in fig. 2.14. Naturally, if a ray does not intersect a bounding volume it cannot intersect any of the contained primitives.

A *bounding volume hierarchy* (BVH) stores the triangles in the leaves of a tree structure with each node corresponding to a bounding volume. Ray traversal can then commence at the root and progresses into the child nodes, in practice, reducing the time complexity to logarithmic in the number of primitives [3, 9].

Other acceleration structures like *binary space partitions* can also be used, but today's consensus is that BVHs are generally the best suited for raytracing, as they guarantee maximum memory usage threshold [3].

2.8   Radiosity

In computer graphics, the term *radiosity* refers to a finite-element approximation of the rendering equation for diffuse reflections. This method has its roots in heat transfer models used in thermal engineering [22] and, in contrast to raytracing, is far more adequate for real-time rendering purposes.

In (2.22) we defined diffuse BRDFs as being constant and irrespective of the angle of observation. This means that, within a static environment, every surface retains its diffuse color no matter what perspective it is being rendered from. Radiosity makes use of this fact by offloading the computation costs required for global illumination to a phase of pre-computation. Pre-computed values are stored in a so-called *lightmap*, then simply retrieved on demand at minimal cost.

This method can only account for diffuse reflections on static (non-moving) geometry. Any displaced object will require the entire generation process to be repeated, while specular reflections have to be deferred to a different method, such as raytracing [23].

The mathematical basis behind radiosity can be derived from the surface-based rendering equation defined in (2.15):

$$L_o(x, \vec{\omega}) = L_e(x, \vec{\omega}) + \int_S f_r(\vec{\omega}', \vec{\omega}, x) L_o(x, \vec{\omega}') V(x, x') \frac{\cos\theta \cos\theta'}{\|x - x'\|^2} dA \qquad (2.32)$$

Since only diffuse reflections are accounted for, the BRDF f_r can entirely be replaced by the constant $\frac{\rho}{\pi}$ defined in section 2.5.3. Furthermore, any directional parameters such as $\vec{\omega}$ can be dropped, as lambertian diffuse reflections are equal in all directions [1]:

$$L_o(x) = L_e(x) + \frac{\rho(x)}{\pi} \int_S L_o(x) V(x, x') \frac{\cos\theta \cos\theta'}{\|x - x'\|^2} dA \qquad (2.33)$$

Similarly to raytracing, radiosity makes use of the *finite element method* by subdividing the environment's geometry into a series of small patches, thus collapsing the area-integral into a finite sum [1]. Fig. 2.15 illustrates the two Riemann sums these approaches correspond to.
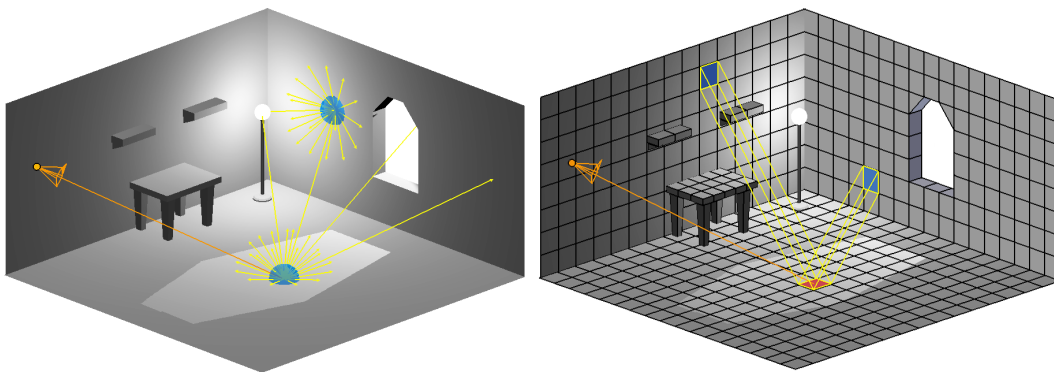
FIGURE 2.15: Illustration of raytracing (left) and radiosity (right), as depicted by Benjamin Kahl [8].

The term $V(x,x')\frac{\cos\theta\cos\theta'}{\|x-x'\|^2}dA$ is separated into a *view factor* $F_{i,j}$ that describes the fraction of the energy leaving patch i that arrives at j. Let n be the amount of surfaces - e.g. patches - in our scene. Then the diffuse radiance of a patch i can be approximated by the following equation [1, 22]:

$$L_o(i) = L_e(i) + \frac{\rho(i)}{\pi}\sum_{j=1}^{n}L_o(j)F_{i,j} \tag{2.34}$$

which constitutes the *Radiosity Equation* as it was formulated in 1984 by Goral et al. [22]. The individual components behind this mathematical transformation is illustrated in fig. 2.16.
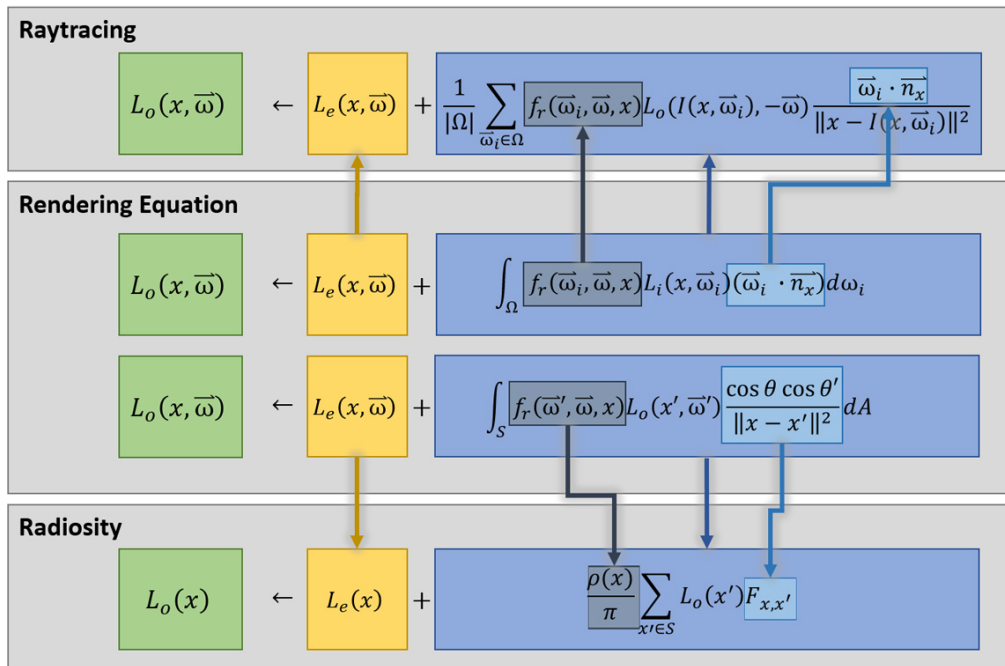
FIGURE 2.16: Approximating the rendering equation through radiosity and raytracing. The inverse square factor is contained within the definition of L_i. Image based on a depiction by Benjamin Kahl [8].

2.8.1 View Factor

Also known as *form factor*, a view factor $F_{i,j}$ describes how well two surfaces i and j are visible to one another and consists purely of geometric parameters.

In the engineering field of *heat transfer*, view factors are calculated through the same geometric term we abstracted from the rendering equation above, integrated for all point-pairs on the two surfaces [69]:

$$F_{i,j} = \frac{1}{A_i}\int_{A_i}\int_{A_j}\frac{\cos\theta_i\cos\theta_j}{\|x_i-x_j\|^2}dA_idA_j \tag{2.35}$$

The division over the surface area A_i results in the *reciprocity rule*, which states that if A_i and A_j are of equal size, then $F_{i,j}$ is equal to $F_{j,i}$ [69]:

$$F_{i,j}A_i = F_{j,i}A_j \tag{2.36}$$

This implies that if both A_i and A_j are known quantities, only half of the form factors need to computed or stored, as each respective mirror pair follows from $F_{i,j} = F_{j,i} \frac{A_j}{A_i}$.

2.8.2 The Nusselt Analog

The numerical computation of form factors is not a wholly simple task, as differential surfaces are difficult to establish. An analog to differential form factors developed by Wilhem Nusselt can provide some useful intuition for the algorithms that follow [1].

The Nusselt analog corresponds to the same procedures outlined in the section on solid angles (see fig. 2.4), where a patch A_j is projected onto an imaginary unit hemisphere centered at A_i and then orthogonally down onto the base of the hemisphere [1, 24]. Thus, the view factor equates to the area projected onto the base divided by the area of the base itself.

The Hemicube Approximation

Nusselt's analog illustrates how two differential surfaces that occupy the same solid angle must have the same form factor. Likewise, if a surface is projected radially onto an intermediate surface, such as a hemicube, the form factor of the projection will be the same as that of the original element [1] (see fig. 2.17).

This is the justification behind the *hemicube approximation*, devised by Cohen et al. in 1985 [24]. It approximates the hemisphere with a hemicube, the faces of which are subdivided into small cells. Once one establishes how many of these cells are occupied by a patch projected onto the hemicube, this results in an amount proportional to the patches' view factor.
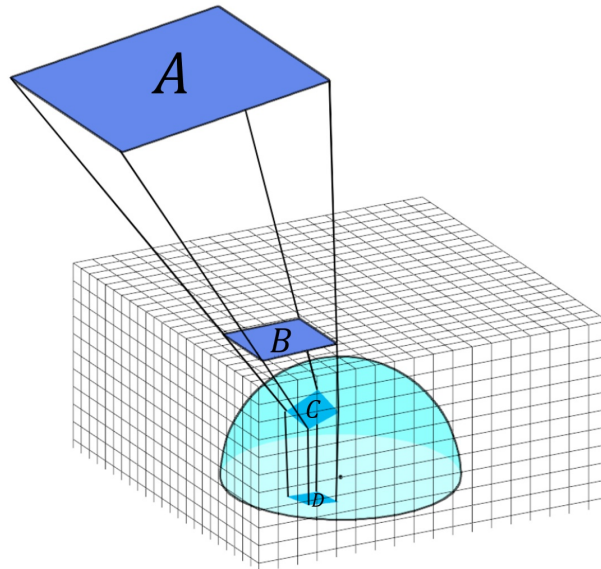
FIGURE 2.17: The justification behind using a hemicube: Patches *A*, *B* and *C* have the same view factor, with *D* corresponding to the Nusselt analog. The size of *B* can be approximated by the amount of cells it occupies on the hemicube. Depiction based on an image by Watt et al. [4].

In graphics card programs the cells of a hemicube can easily be encoded as pixels on a *cubemap*. Figuring out which cells a surface occupies then simply amounts to rasterizing said surface into a cubemap from the perspective of the patch (see fig. 2.18). Rasterizing all other surfaces alongside it lets us make a full accounting of which surface has what contribution (view-factor) from the current patch, including the visibility term *V*. This process is closely related to *Z-Buffering*, which is described in further detail in section 2.9.1.
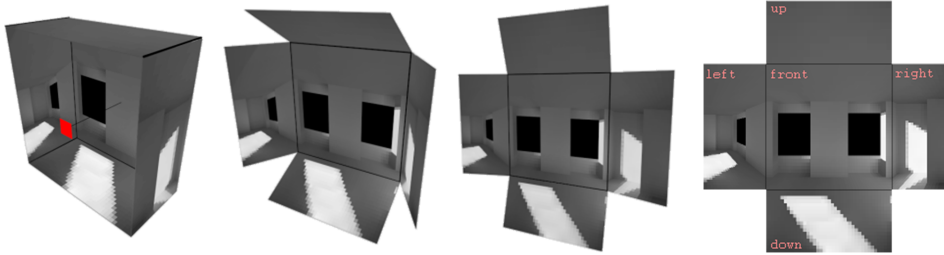
FIGURE 2.18: Rendering a scene through a hemicube to determine visible patches and their view factors, giving an overall approximation of irradiance. Image adapted from work by Hugo Elias [77].

Monte-Carlo Integration

An alternative method of determining a view factor is to simply use a randomized sample set of pairs that are uniformly distributed points from each surface. Let x_{ki} and x_{kj} be the k'th random pair of points for the two surfaces A_i and A_j respectively. Then the form factor between A_i and A_j can be computed through a Monte-Carlo approximation of K samples [25] (see fig. 2.19):

$$F_{i,j} = \frac{1}{K} \sum_{k=1}^{K} A_j \frac{\cos \theta_{ki} \cos \theta_{kj}}{\|x_{ki} - x_{kj}\|^2} V(x_{ki}, x_{kj}) \tag{2.37}$$
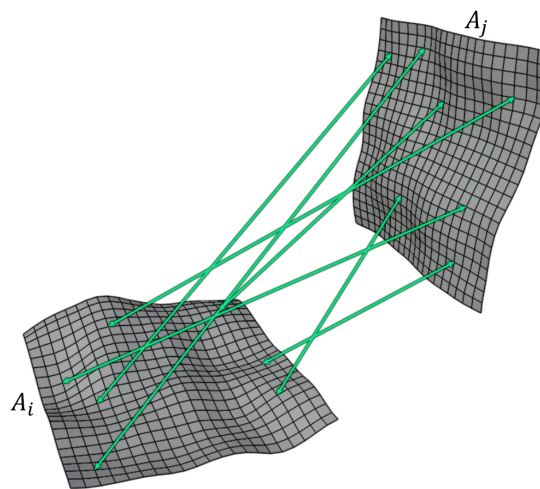
FIGURE 2.19: Form factor calculation requires solving a double integral over surfaces of patches. We can instead randomly sample the 4D space that each patch-pair lies in to get an accurate estimate.

2.8.3 Classical Radiosity

Recall the radiosity equation as defined in (2.34):

$$L_o(i) = L_e(i) + \frac{\rho(i)}{\pi} \sum_{j=1}^{N} L_o(j) F_{i,j} \tag{2.38}$$

Now, where the view factors $F_{i,j}$ have been defined, the radiance values L_o can be formulated as a solution vector, which allows the problem to be entirely expressed as a matrix equation [1, 69]:

$$\begin{bmatrix} L_o(1) \\ L_o(2) \\ ... \\ L_o(n) \end{bmatrix} = \begin{bmatrix} L_e(1) \\ L_e(2) \\ ... \\ L_e(n) \end{bmatrix} + \begin{bmatrix} \rho_1 & 0 & ... & 0 \\ 0 & \rho_2 & ... & 0 \\ ... & ... & ... & ... \\ 0 & 0 & ... & \rho_n \end{bmatrix} \begin{bmatrix} F_{11} & F_{12} & ... & F_{1n} \\ F_{21} & F_{22} & ... & F_{2n} \\ ... & ... & ... & ... \\ F_{n1} & F_{n2} & ... & F_{nn} \end{bmatrix} \begin{bmatrix} L_o(1) \\ L_o(2) \\ ... \\ L_o(n) \end{bmatrix} \tag{2.39}$$

Through some algebraic transformation (see Cohen et al. [1]), this can formally be written as:

$$L_o = (I - \rho F)^{-1} L_e \tag{2.40}$$

where I is an identity matrix of size $n \times n$.

Solving this system yields the complete solution to the radiosity equation directly, but requires the entire computational cost to be paid upfront, which becomes prohibitive for larger values of n. Instead, it is common practice to solve the equation progressively, with each bounce of light performed separately [70].

2.8.4 Progressive Radiosity

The nature of diffuse reflections combined with the inverse square law implies that lighting values converge rather quickly, which can be leveraged in iterative solutions where each iteration applies the calculations for a single bounce of light. The amount of iterations - e.g. passes - will determine the brightness and fidelity of the scene but also linearly impact the required computation time. Since each subsequent bounce has a lesser impact on the resulting image, fewer than 16 iterations tend to be sufficient in most cases, after which the difference in radiance tends to become non-tangible.
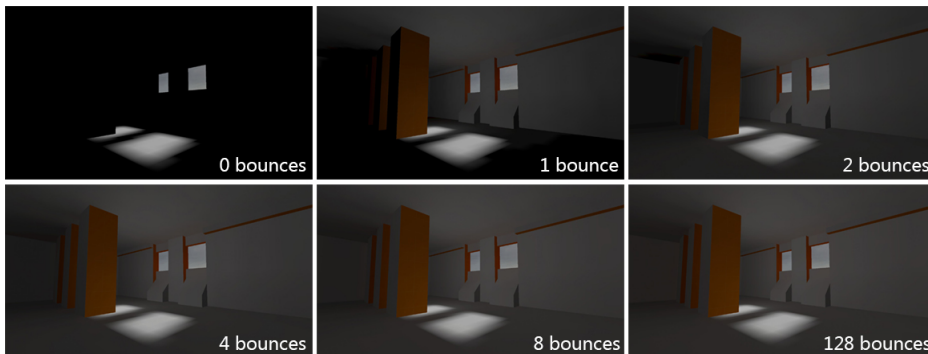

FIGURE 2.20: Progressive radiosity after a specified amount of additional light bounces, as done by the 2014 VRAD tool and rendered by the 2009 Source Engine. Image from work by Benjamin Kahl [8].

The prevalence of algorithms such as *instant radiosity* [26] and *voxel cone tracing* [27] imply that, frequently, a mere two bounces tend to be sufficient for an adequate approximation of indirect light (see fig. 2.20). Progressive radiosity can be halted after any iteration, once a desired solution has been reached.

Standard iterative methods for solving matrix equations include the *Jacobi iteration* and the *Gauss-Seidel method* [69]. The solution can also be configured as a *shooting* or *gathering* variant, depending on which patches are processed in the algorithms outermost for-each loop [28].

A generalized pseudo-code of progressive radiosity (adapted from Wüthrich [70]) is listed below:

Algorithm 1 Progressive Radiosity

1: **for** each iteration **do**
2:     **for** $A_i \in S$ **do**
3:         **for** $A_j \in S$ **do**
4:             Calculate or retrieve $F_{i,j}$
5:             Update radiosity of A_j
6:             Update emission of A_j
7:         **end for**
8:         Set emission of A_i to 0
9:     **end for**
10: **end for**

Henceforth, we will refer to this algorithm as *pure* progressive radiosity, to differentiate it from its variants that employ additional enhancements to improve performance.

2.8.5 Progressive Refinement Radiosity

An extension to pure radiosity, *progressive refinement radiosity*, was first introduced by Cohen et al. in 1988 [29]. This reformulation of the original algorithm eliminates the memory requirements for view factors entirely by computing them on-the-fly. Patches are processed in sorted order according to their energy contribution to the environment and then updated simultaneously after each pass.

More importantly is the use of *refinement* through adaptive subdivision, which had already been introduced by Cohen et al. in 1986 [30]. This process dynamically sub-divides or merges individual radiosity patches depending on the gradient across them. The resulting *quad-tree* will have more leaves in places of relevance, such as the boundary of a shadow, whilst treating, flat mono-colored surfaces as single patches (as depicted in fig. 2.21).

Refinement in general is not exclusive to progressive radiosity and can be performed in a number of varieties. Most techniques operate *a posteriori*, meaning they adjust the amount of patches based on the output of each iteration. These are commonly categorized as follows (as done by Slusallek et al. [69]):

- *r-refinement:* Repositions vertices of a mesh based on the lighting gradient.

- *h-refinement:* Stores lighting data in a quad-tree and subdivide each node depending on a gradient threshold.

- *p-refinement:* Increases polynomial order of patches depending on a gradient threshold.
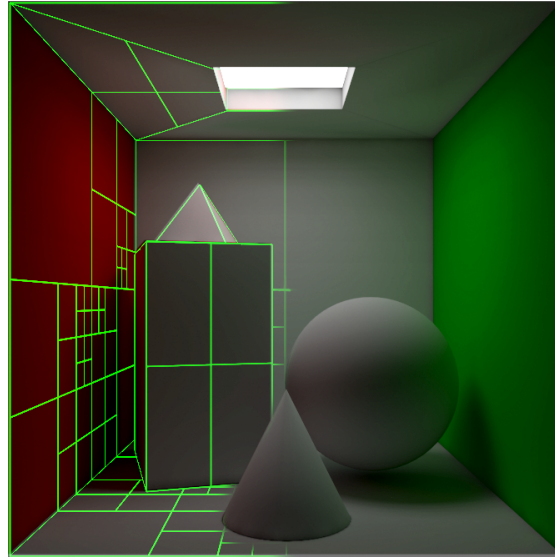
FIGURE 2.21: Radiosity with adaptive subdivision. More patches are
allocated to areas with a high gradient, such as shadow boundaries.
Image based on a concept by Coombe et al. [17].

- *remeshing:* Re-computes an entirely new mesh, with edges and vertices aligned
  along shadow boundaries.

The primary goal behind refinement is to drive down the need for large patch
amounts, thus improving performance.

Some solutions (see Coombe et al. [5]) model the patches that are sampled (i.e
shot towards) as separate sub-set of the set of all patches in scene. This allows one
to maintain a large set of evenly distributed patches, whilst only sampling the most
important ones, in accordance to a subdivided quad-tree.

2.8.6 Instant Radiosity and Sampling Approaches

To complement radiosity's slow rate of convergence and static geometry constraints,
Instant Radiosity was introduced by Keller in 1997 [26]. It approximates global illumi-
nation effects by creating additional, *virtual light-sources* on inter-reflecting surfaces,
making it perfectly fit to be used within real-time requirements [78].

Rays are shot in random directions from light-sources. Then, at their intersection
locations with other geometry, *virtual point lights* (VPLs) are created that emit light
corresponding with the underlying surface's color and brightness (see fig. 2.22). The
process can be repeated for each of these light-sources recursively, with the amount
of virtual point-lights ultimately determining the quality of global illumination.

Instant radiosity does not require costly pre-computations and can accommo-
date dynamic scene changes on demand, but generally does not produce the same
lighting and shadow quality as regular radiosity, and requires significant amount of
GPU power to run in real-time. *Incremental* instant radiosity [31] allows incremen-
tally adding new VPLs over time, whilst maintaining an even distribution of VPLs
across the scene.

Instant radiosity can loosely be classified as a *directional sampling* approach. In
regular radiosity, visibility is determined separately for all possible patch-pairs, which
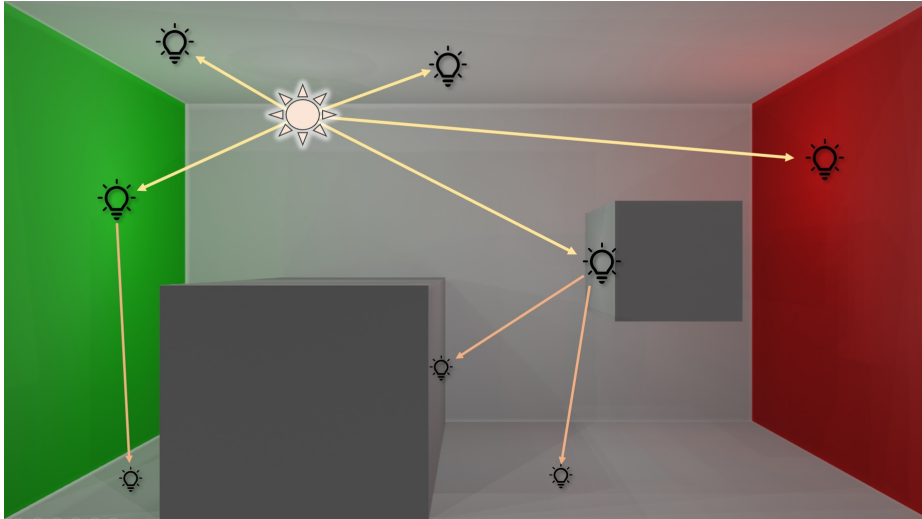
FIGURE 2.22: Concept behind instant radiosity. Rays are scattered from a light-source. On their points of collision we place a new, virtual point-light. The entire scene (including virtual point-lights) can then be rendered using a model like phong.

results in a complexity of at least $O(n^2)$ in the number of patches. In directional sampling approaches, we isotropically scatter a set of rays for each patch, then calculate lighting contribution for each patch hit by a ray. In theory this drops the complexity to $O(n * m)$, where m is the maximum amount of samples taken for each patch.

2.9 Visibility Determination

The surface-based rendering equation in (2.15) defines a function $V(x, x')$ that is equal to 1 if no other geometry lies between the two points x and x', otherwise 0.

Determination of visibility has been a cornerstone problem in computer graphics from its very beginning. Conundrums such as the *art gallery problem*, *watchman route problem* or graph visibility provide great insight into the theoretical perspective in that the underlying issue is NP-hard [32].

A common solution in computer graphics is to leverage the celerity of rasterization pipelines to calculate approximate visibility through a Z-Buffer. This method is frequently employed for the calculation of shadows, whilst a binary-space partition (BSP)[2] restricts the selection to only relevant geometry [33].

2.9.1 Z-Buffering

A z-buffer (or depth buffer) consists of the clip-space z-coordinates for every pixel of a rendered primitive. These values correspond to the "image depth" or the distance of the painted geometry to the camera [67] (see 2.23 for an example). Z-buffers are regular by-products of the rasterization process.

To determine whether an object is visible to a certain location, a scene can be rasterized from the perspective of the given point, as described by the hemicube approximation (see 2.8.2). Instead of the object's colors, it is sufficient to simply store a unique ID for each object into the pixels of the raster image. The result will contain

---

[2]Sometimes referred to as Portal-Engine / Portal rendering
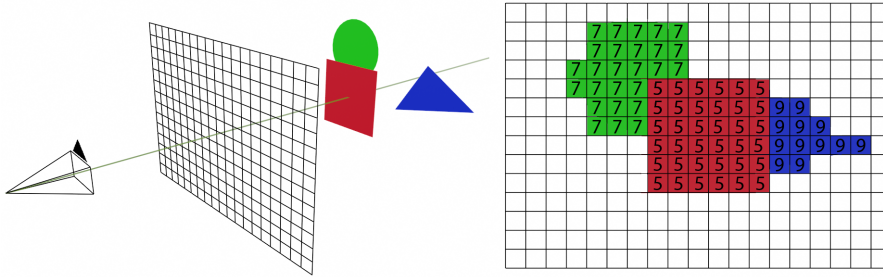
FIGURE 2.23: Rasterization of basic shapes (left) and respective z-buffer (right).

the IDs of all visible objects, barring translucent or small surfaces that occupy less than a pixel [67, 34].

This method is frequently used to generate shadows in real-time, although a low render resolution can impair the quality and can lead to shadow pixelation (see fig. 2.24).
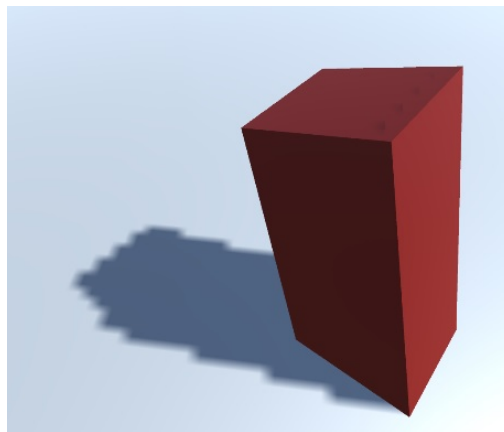
FIGURE 2.24: Red cuboid with a pixelated shadow as a result of z-buffering (Unity Engine 2021.3.21f).

2.9.2 Raytracing for Visibility

Instead of translating a 3D location to a screen position, we can translate a screen position to a 3D location by tracing a ray through the corresponding pixel in the image plane. Likewise, two locations x_1 and x_2 are mutually visible if a ray launched from x_1 can reach x_2 unimpeded:

$$V(x_1, x_2) \leftrightarrow I(x_1, x_2 - x_1) = x_2 \tag{2.41}$$

Whilst z-Buffering can be very quick in computing the visibility of several objects from a single location (like with shadows of a point-light), it is not as well-suited for the visibility computation of large sets of arbitrary point-pairs, as is required by radiosity.

Chapter 3

The Turing Architecture and DXR

In August 2018 the multinational tech company *Nvidia* introduced the first consumer products capable of genuine real-time raytracing in the form of the *GeForce 20 series* of GPUs.

Built on a newly developed *Turing microarchitecture*, these chips subsume several different types of specialized processor cores to accelerate their respective tasks considerably [49]. The so-called *RT core* is designed specifically to process the traversal of bounding volume hierarchies and triangle intersection tests, thereby enabling Turing GPUs to execute rudimentary raytracing algorithms at a rate of several billions of rays per second [49, 3].

In this chapter we will review the exact makeup of this architecture and what its key enablers are for the significant boost in graphics performance. Additionally we will outline its embedded solution for real-time raytracing and how it can be used through the DirectX 12 raytracing pipeline.

3.1 GPUs and Parallelism

GPUs (graphics processing units) can be generally defined as specialized processing units designed for the quick computation and management of visual data in a frame buffer.

In essence, their primary intended task is to continuously compute 2D matrices of color values that represent pixels on a screen. For most intents and purposes (excluding post-processing effects such as anti-aliasing, bloom or blurring), the color of any one individual pixel is independent of its predecessor and neighbours. As such, this task is highly adequate for parallelization.

Fig. 3.1 shows the generalized differences in architecture between CPUs and GPUs: The latter contain hundreds to thousands of cores/ALUs (arithmetic logical units) that can run a large set of lightweight threads simultaneously, whilst CPUs are geared towards a small number of highly efficient, general-purpose cores and fast access to system memory.

Due to their immense power in parallel computing, GPUs have found use in many other areas than graphics, such as machine learning and cryptography.

3.2 The Turing Architecture

The Turing GPU microarchitecture (named after *Alan Turing*) was introduced in August 2018 [49] and is the architecture that Nvidia's newest range of consumer-grade graphics processors are built upon.

Turing cards inherit large parts of their design from earlier microarchitectures, namely the 2010 *Fermi* [52] architecture and its successors (*Kepler (2012)*, *Maxwell*
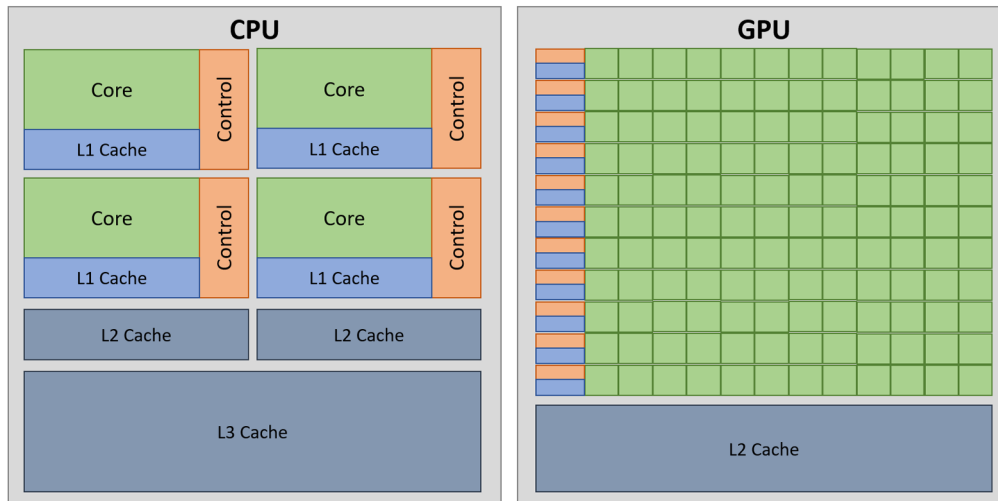
FIGURE 3.1: General architecture of a CPU (left) vs. a GPU (right). Image adapted from the Nvidia CUDA programming guide 2022 [51].

(2014) and *Pascal (2016)*). Turing's key enabler for its new features and improved performance is the overhauled GPU processor which accommodates "improved shader execution efficiency, and a new memory system architecture that includes support for the latest GDDR6 memory technology"[49].

3.2.1   TU102 GPU Structure

The TU102 is a a high-end GPU of the GeForce 20 series and is divided into six Graphics Processing Clusters (GPCs) each of which contains a hybrid makeup of computational units [49]. Below we provide a description of its primary components in line with the *Turing architecture whitepaper* [49].

The GPU receives its instructions and data through a PCIe 3.0 interface, which connects it to the rest of the computer and acts as the main access point to the systems main memory [49]. Each GPC comes equipped with its own L1 cache, whilst all six GPCs share twelve memory controllers and an additional 512kb of L2 cache [49].

An onboard, chip-level *GigaThread Engine* receives command queues from the host via the PCIe bus and executes these by setting up shaders on available hardware [52]. This is the central command processor that manages the entire chip, including context switches, scheduling and power management [52].

3.2.2   GPC

Each GPC, as depicted in fig. 3.2, houses six *Texture Processor Clusters* (TPCs), which are nested sub-clusters in of themselves, as well as a dedicated raster engine [49]. The GPC/TPC subdivision is useful for distributing and allocating of the computational workload evenly, as they can be managed as a single unit [49].

Each TPC consists of two Streaming Multiprocessors (SM), which the parent TPC can wake and sleep based on how heavy the incoming workload is [49].

FIGURE 3.2: Architecture of the TU102: 6 GPCs, each consisting of 6
TPCs that contain 2 SMs each. Each SM contains a number of ALUs
for integer, floating point and matrix operations as well as a single RT
core. Each RT core has a separate core for BVH traversal and triangle
intersections respectively. Image adapted from the Turing architec-
ture whitepaper [49].

3.2.3 SM - Streaming Multiprocessor

The streaming multiprocessor houses all the primary computation units of the GPU
itself. Each one contains its own onboard memory in the form of a 256KB register
file, four texture units and 89 KB of L1 shared memory which can be dynamically
allocated depending on the workload [49].

Each SM contains three different types of processing cores that represent the
GPU's entire computational capacity [49]:

- 64 CUDA cores (4608 total)

- 2 Tensor cores (576 total)

- 1 RT core (72 total)

All of these components can be identified on the GPU die itself, as shown in fig. 3.3.
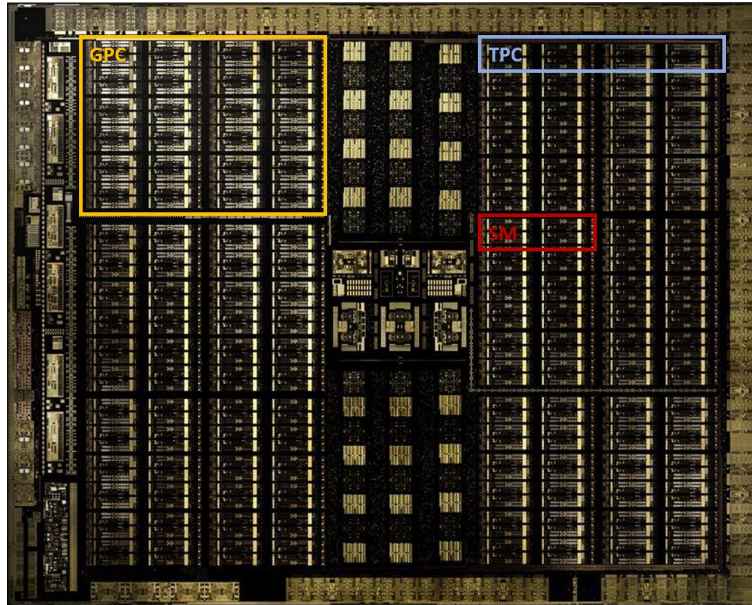
FIGURE 3.3: Turing GPU die. Image adapted from the Turing architecture whitepaper [49].

CUDA Core

CUDA cores are basic computational units that perform common integer- or floating point operations [49, 51]. These can be used both for rendering but also other parallel computing purposes through the CUDA platform[1].

Unlike its predecessors, the Turing architecture provides separate data-paths for integer and floating-point operations. In previous generations the execution of an integer-based instruction would have blocked floating-point instructions from issuing [49].

Tensor Core

Tensor cores are specialized execution units that are built specifically and purely to accelerate the process of matrix (or tensor) multiplication [79]. These find heavy usage in vertex transformation operations of common rendering applications but also for deep learning (neural network) purposes.

Turing's tensor cores have been enhanced for inferencing and equipped with additional integer precision modes [49].

---

[1]Note that we are using the term "CUDA core" as an umbrella term for a GPUs general purpose ALUs. Whilst CUDA is technically an Nvidia-exclusive platform, all GPUs are equipped with equivalent computational units.

RT Core

One of the biggest innovations of the Turing architecture is the introduction of *RT cores*, a specialized processing core that exclusively performs BVH traversal and ray-triangle intersections to facilitate real-time raytracing [49]. The underlying acceleration concepts are not new; parallelization and BVHs have been common practice in raytracing for some time.

But unlike previous raytracing solutions built on the GPU, RT cores can run autonomously, in parallel to the CUDA cores and thus offload the SM [49]. This means that when a shader dispatches a ray, it can continue performing other calculations whilst the ray is being traced in parallel. Similarly, once a ray intersects geometry, the RT core can directly move on to tracing the next ray, whilst the CUDA and tensor cores take care of shading and lighting. The horizontal parallelism this concept provides is demonstrated in fig. 3.4.

RT cores consist of two different units: one for bounding box testing and a second for ray-triangle intersections [49]. This distinction allows for further horizontal parallelism, as the first unit can move on to the next triangle/input whilst the second unit still finishes the previous input.
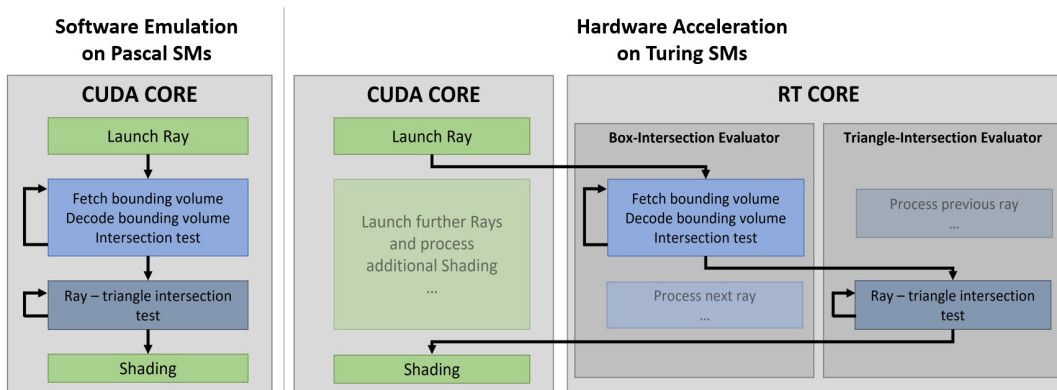
FIGURE 3.4: Software emulated raytracing on Pascal GPUs (left) vs. hardware accelerated raytracing on Turing GPUs (right). Image in line with the depictions in the Turing whitepaper [49].

Pascal GPUs have retroactively been made compatible with RTX through the DirectX raytracing API. But since no RT cores are available on these, a software approximation that runs on CUDA cores is employed instead (see fig. 3.4). This results in significant performance degradation. Turing GPUs can process 10+ Giga-Rays per second, whilst high end Pascal GPUs only reach approximately 1.1 Giga-Rays per second [49].

The general umbrella term for Nvidias real-time raytracing technology is *RTX*, which is an abbreviation for *Ray Tracing Texel eXtreme*.

3.3 DirectX

Rendering applications typically interact with GPU hardware through graphics APIs that span across multiple programming languages and platforms. These describe an abstract programming layer that specifies exactly what result, input and output of each function ought to be. Minor variations in how computation is performed on a

hardware-level, can produce divergent results even when a program is executed on the same API[2].

The most notable APIs are *OpenGL*, *Vulkan* and *DirectX*, which have all seen industry-wide adoptions across most common graphics cards and applications [10].

Upon their release, Turing's raytracing features were only exposed through DirectX 12, later becoming available on Vulkan and (partially) OpenGL as well [54]. Since the compatibility has most matured on DirectX, the remainder of this thesis will focus primarily on DirectX 12.

3.3.1 DirectX Rasterization Pipeline

When a scene is rendered into a frame output buffer using rasterization, DirectX employs a programmable graphics pipeline, which sequentially executes a series of highly specialized, fixed-function steps that allow the system to efficiently draw abstract 3D geometry in a given perspective. Each of these steps executes a concrete task with the output of the previous step as its input parameters. Given the consecutive nature of the pipeline, the individual steps can easily be executed in parallel for successive frames to be rendered.

The GPU programs that can be inserted in-between these steps are commonly termed *shaders* and let developers configure custom pipeline behaviour. In a DirectX context, shaders are written in a *high-level shader language* (HLSL), which strongly resembles the syntax used in C-based languages.

Fig. 3.5 shows a simplified overview of the individual stages that form the DirectX rasterization pipeline (as described in the DirectX 12 documentation [50]).
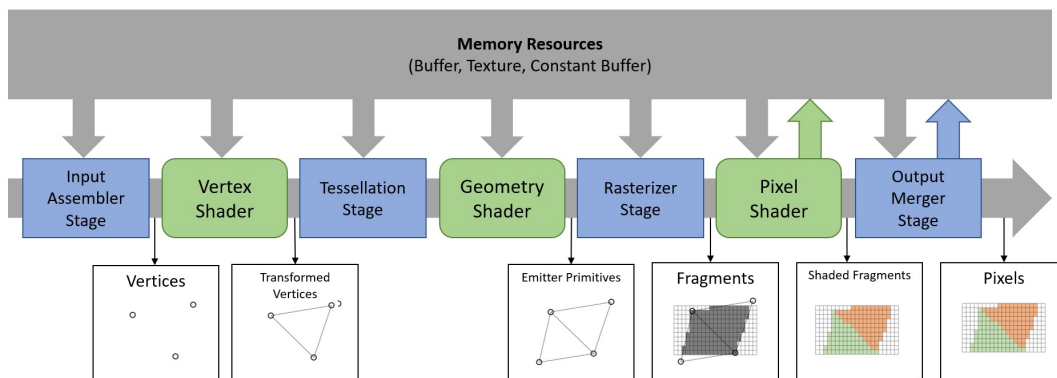
FIGURE 3.5: Stages of the DirectX 12 rasterization pipeline. Programmable shaders are marked in green. Image based on information from the DirectX12 documentation [50].

Input Assembler Stage

The Input Assembler Stage reads geometry data from the allocated buffers and assembles it into primitives (usually triangles) that are usable by the other pipeline stages [50]. This stage will also attach system-generated values, such as primitive

---

[2]For instance, *Qualcomm Adreno 2xx* processors use 24 bit floating precision in fragment shaders, whilst the Nvidia X1 uses the more common 32 bit precision. The OpenGL 4.6 specification states that it "does not guarantee an exact match between images produced"[53].

IDs, instance IDs or a vertex IDs, so that subsequent shader stages can reduce processing to only instances, primitives or vertices that have not already undergone processing [50].

The primitives generated by the input assembler stage are subsequently transferred to the vertex shader.

Vertex Shader

The vertex shader is the first and arguably most important geometry processing step in any graphics pipeline. It is an input-output program that is executed on every vertex individually and lets the user transform, modify or otherwise set up vertex-specific data for later pipeline stages [75].

The most common operations performed in the vertex shader are the applications of the model-, view- and projection- matrices [67, 75]. To conserve memory, 3D model data is stored in local coordinates so each instance of any 3D model in a scene comes with a model matrix that describes the object's position, rotation and scale. The model matrix is a 4x4 affine transformation matrix that converts vertex positions from local space into world space.
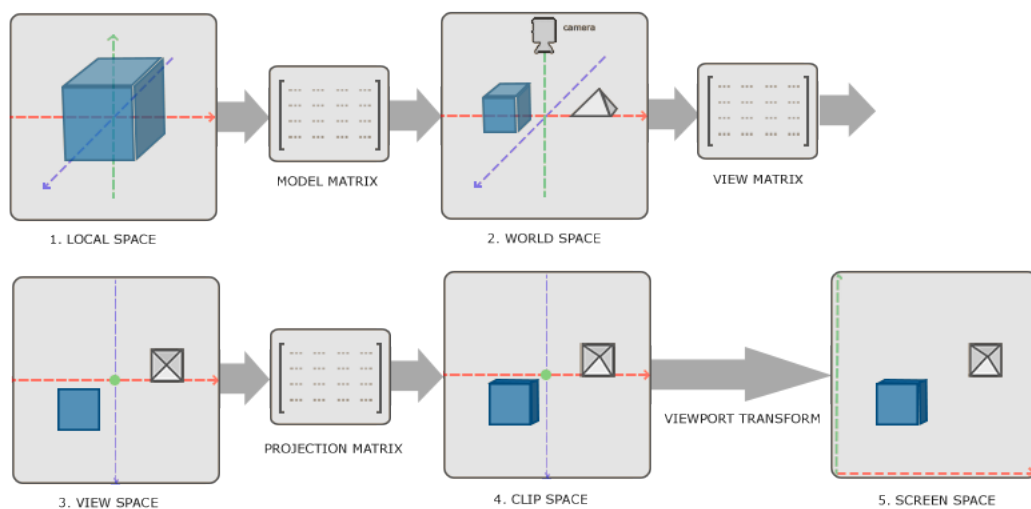
FIGURE 3.6: Coordinate system transformations in the vertex shader, as pictured by Joey de Vries [75].

The view matrix will thereafter transform the coordinates into view space and the projection matrix applies the FOV-based distortion of non-orthographic cameras to each vertex. The final vertices find themselves in a $[-1, 1]$-ranged clip space coordinate-system, where vertices outside this range lie outside the camera's view frustum. The entire transformation process is depicted in fig. 3.6.

Furthermore, additional data-points such as a vertex's texture coordinates and normal vector are likewise set up in the vertex shader. By the end of this stage, each vertex will possess its own set of data known as *attributes*, which subsequent pipeline stages linearly interpolate on for respective values anywhere on the spanned primitive.

Tessellation Stage

The tessellation stage is an optional stage added in DirectX 12, which allows the user to generate additional vertices directly on the GPU [50]. The utility of this stage plays no importance in remainder of this thesis.

Geometry Shader

Another optional step that allows for additional processing is the geometry shader, which is executed on a per-primitive basis. For triangles, each set of three vertices relayed by the vertex shader are passed into the geometry shader as a triplet, which lets the user remove, subdivide or otherwise transform them in a manifold of ways [75, 50].

This stage is frequently used for triangle-based effects like enlargements or shrinking (see fig. 3.7) as well as a vital part for GPU-based voxelization [8, 71] (see 5.3.2).
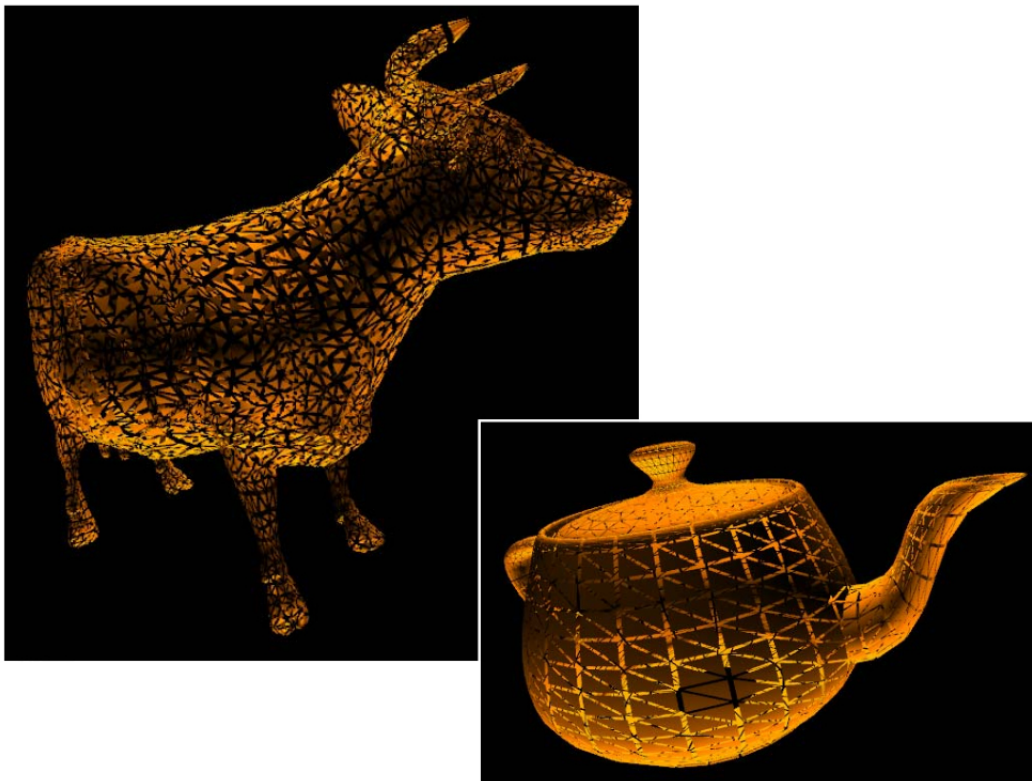
FIGURE 3.7: Example: Shrinking triangles in the geometry shader. Image by Bailey [72].

Rasterizer Stage

As implied by its name, the rasterizer stage converts each primitive into the set of pixels that it occupies on-screen, whilst interpolating the per-vertex attributes across it, so that each pixel has a corresponding set of values for normal vectors, positions, texture coordinates etc.

If the pipeline is set to utilize multi-sampling (compute several color values per pixel), the individual sub-samples are likewise arranged here [50]. This stage additionally discards any pixels that do not face the camera (face culling), are occluded

by other objects (depth clip) or are outside the viewport (scissor clip) [50]. This limits the amount of pixels - i.e. fragments - that need to be processed in the next stage.

Pixel Shader

The pixel shader (known as *fragment shader* in OpenGL) is executed once for each pixel provided by the rasterization step [75, 50]. This is typically the most performance-intensive stage, as it is executed most frequently and includes the vital lighting calculations that produce the final color of each pixel.

Mathematical models like *Phong* [20] can be used to compute the pixel brightness and color composition equating to the radiance emanated from the fragment position towards the camera. The input data for these will be a position-based linear interpolation between the vertex attributes of each triangle. The factors of this interpolation can be equated with the fragment's barycentric coordinates on the triangle.

Output Merger Stage

The final stage of the pipeline utilizes any present depth/stencil buffers to perform depth-testing and blending of transparent objects (alpha testing) with the colors provided by the pixel shader to generate the final image [50].

If the image is being rendered into multiple render targets (DirectX 12 supports up to 8 [50]), the writing process is likewise handled by the output merger stage.

3.4 DirectX Raytracing

Since the raytracing procedure fundamentally differs from classic rasterization, DirectX introduces an entirely new graphics pipeline with a whole new set of programmable shaders to accommodate it.

DirectX Raytracing, or simply *DXR*, shares several similarities with its rasterization counterpart, as it aims to strike a balance between fixed-function and programmable stages to maximize both execution efficiency and potential for customization [55].

3.4.1 DXR Pipeline

The system is intended to process rays independently and in parallel. Once a ray hits or misses, it can create further sub-rays, but generated rays that are in-flight can never be dependent on each other [55].
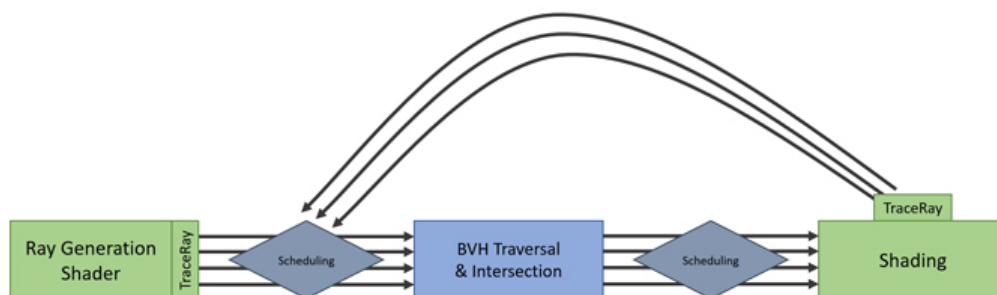

FIGURE 3.8: Simplified overview of RTX pipeline stages. Depiction based on an image from the Microsoft DXR Specification [55].

On a surface level, the raytracing pipeline can be divided into three core components: Ray generation, BVH traversal and shading (see fig. 3.8). Scheduling functionalities of which ray or shader is processed in which order is an opaque, hardware-bound process and cannot be altered [55]. Likewise, the BVH traversal is treated as a single, fixed-function step that is offloaded to the RT cores. In their examination of RTX performance, Sanzharov et al. [35] concluded that RTX does some ray grouping and sorting during the GPU work creation process, in order to speed up bundles of rays traversing through the same BVH leaves.

Rays in DXR

The data structure that represents a ray in DXR closely follows the definition put forth in section 2.7.1, consisting of an origin, direction and a min-max distance interval [3]. Each of these values needs to be initialized before the ray can be traced:

```
1  struct RayDesc
2  {
3      float3 Origin;
4      float3 Direction;
5      float TMin;
6      float TMax;
7  };
```

Furthermore, rays in DXR can carry a *payload*, which is a custom data-structure of limited memory that can be accessed on a per-ray basis by any of the programmable shaders [3]. An example of how this payload may be utilized is given in section 3.4.3.

Programmable Shaders

The DirectX raytracing pipeline introduces a total of five new programmable shader types that are invoked during the raytracing process based on the flow diagram depicted in fig. 3.9.

Below we list a brief description for each one in line with how their documentation in *Ray Tracing Gems* [3].

- The *ray generation* shader is launched once for every instance of some enumerable index (1D, 2D or 3D grid) and handles the initial ray launches [3]. In a traditional raytracing application, this shader would launch the initial rays from the camera through each virtual pixel. An HLSL `TraceRay(...)` function is available for this purpose.

- *Intersection shaders* define how intersections are calculated with arbitrary primitives [3]. By using intrinsic functions like `ReportHit()` or `AcceptHitAndEndSearch()` the user can define which intersections are counted as hits or not. If no intersection shader is provided, the pipeline employs a high-performance default that uses triangles [3]. Utilizing intersection shaders instead of the build-in ray-triangle intersection is less efficient but offers far more flexibility [3, 55].

- As the intersection shader defines which hits to report and which not to, *any-hit shaders* are executed for all reported hits [3]. In addition to running regular
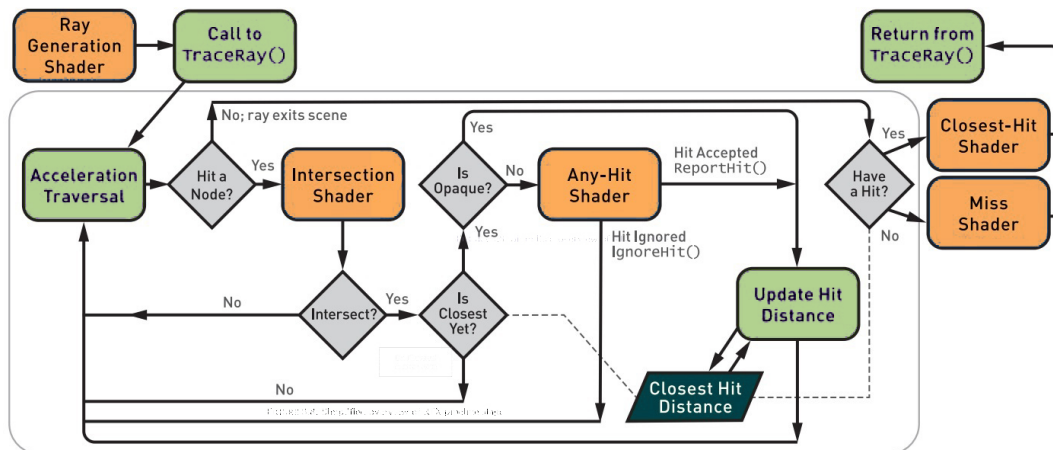
FIGURE 3.9: Overview of the DXR pipeline as depicted in the *Raytracing Gems* book [3]. It includes the five programmable shader types marked in orange. BVH traversal is marked by a grey outline.

HLSL code (like writing data into textures on hit) any hit shaders allow otherwise valid intersections to be discarded, such as transparent surfaces.

- As the name implies, *closest-hit shaders* are executed at the closest intersection for each ray [3]. In a traditional raytracing context, this would recursively launch additional rays to compute the color the hit location has.

- If no hit is registered for a ray, the *miss shader* is executed [3]. This can be used, for instance, to display a background color for pixels not occupied by geometry.

This pipeline shares many components and concepts with regular raytracers. Algorithm 2 illustrates how a standard DXR pipeline would function if it were executed in sequence.

Algorithm 2 DXR Raytracing Process (Adapted from Raytracing Gems [3])

1: **for** $x, y \in image.dimensions()$ **do**
2:     $ray \leftarrow createRay(x, y)$ ▷ Ray from C through pixel (x, y)
3:     $closestHit \leftarrow null$
4:     **while** $leaf \leftarrow findBvhLeadNode(ray, scene)$ **do**   ▷ BVH traversal
5:         $hit \leftarrow intersectGeometry(ray, leaf)$ ▷ *intersection* shader
6:         **if** $isCloser(hit, closestHit)$ & $isOpaque(hit)$ **then**
7:             $closestHit \leftarrow hit$
8:         **end if**
9:     **end while**
10:     **if** closestHit **then**
11:         $image(x, y) \leftarrow shade(ray, closestHit)$ ▷ *closest-hit* shader
12:     **else**
13:         $image(x, y) \leftarrow miss(ray)$ ▷ *miss* shader
14:     **end if**
15: **end for**

3.4.2 TraceRay Function

The DXR-intrinsic `TraceRay(...)` function can be used in programmable shaders to commence a raytracing process on an RT core. It operates under the following parameters [55]:

```
1  TraceRay (
2      RaytracingAccelerationStructure  AccelerationStructure ,
3      uint  RayFlags ,
4      uint  InstanceInclusionMask ,
5      uint  RayContributionToHitGroupIndex ,
6      uint  MultiplierForGeometryContributionToHitGroupIndex ,
7      uint  MissShaderIndex ,
8      RayDesc  Ray ,
9      inout  payload_t  Payload
10  ) ;
```

- The first parameter selects the BVH containing the geometry that needs to be traced. In most instances only one BVH of a scene exists, but multiple ones can be defined to trace on different scenes from a single shader.

- The second parameter is an integer where each bit represents a certain flag that affects ray behaviour. Some notable flags are

  `RAY_FLAG_CULL_BACK_FACING_TRIANGLES` to disregard intersections on triangles not facing the ray, or

  `RAY_FLAG_ACCEPT_FIRST_HIT_AND_END_SEARCH` to terminate the BVH traversal immediately when any geometry is hit.

- The third parameter is an instance mask that allows skipping geometry on a per-instance basis. Passing a value of `0xFF` (in hexadecimal) would cause all geometry to be tested for intersections, whilst `0x00` would test none.

- The fourth and fifth parameters define the hit-group for this ray. A hit group consists of an intersection, closest-hit and anyhit shader. Typically each ray-type is separated into a respective hit-group so that the respective shader code is executed. (For instance, shadow-rays, diffuse reflections and specular reflections would each possess their own hit groups)

- The sixth parameter lets us configure which miss shader to use, irrespective of the hit group.

- The seventh parameter is the ray description, as defined in 3.4.1, and

- the eighth parameter is the payload the ray carries over its lifetime.

An alternative `TraceRayInline(...)` function exists, that does not use separate shaders, and defers all shading to the caller [50].

3.4.3 Code Example

The code below serves as a simple example for the respective shaders a simple DXR program for shadow rays would utilize. The respective anyhit and miss shaders are marked by [shader("anyhit")] and [shader("miss")] attributes respectively:

```
1   // Custom ray−payload datatype
2   struct ShadowPayload {
3       bool isVisible;
4   };
5
6   [shader("miss")]
7   void ShadowMiss(inout ShadowPayload payload) {
8       payload.isVisible = false;
9   }
10
11  [shader("anyhit")]
12  void ShadowAnyHit(intout ShadowPayload payload,
        BuiltInTriangleIntersectionAttributes attrib)
13  {
14      // Ignore transparent surfaces
15      if( isTransparent(attrib, PrimitiveIndex() ) ) {
16          IgnoreHit();
17      }
18  }
19
20  [shader("raygeneration")]
21  void rayGen()
22  {
23      float3 origin = CameraPosition();
24
25      float3 dir = PixelPosition(DispatchRaysIndex().xy) −
          origin;
26
27      RayDesc ray = {
28          Origin = origin,
29          Direction = dir,
30          TMin = 0.001f,
31          TMax = 1000f
32      };
33
34      ShadowPayload payload = { true };
35
36      TraceRay( scene,
37              RAY_FLAG_SKIP_CLOSEST_HIT_SHADER,
38              0xFF, 0, 1, 0, ray, payload );
39
40      if( payload.isVisible ) {
41          OutputTexture[DispatchRaysIndex().xy] = RED;
42      }
43      else {
```

```
44            OutputTexture[DispatchRaysIndex().xy] = BLACK;
45        }
46  }
```

This example colors every pixel that is occupied by geometry in red. It uses the ray payload to mark whether a pixel is occupied by any non-transparent geometry. The isVisible component is initialized as true, but is set to false in the miss shader. The anyhit shader discards intersections with transparent geometry [3].

3.4.4 Host Initialization

The foregoing sections provide a comprehensive overview of the DXR raytracing pipeline and its GPU-side functions and capabilities. However, as with any graphics API, the global pipeline state and execution is managed by the device host (CPU side) through a series of API function calls.

Low level DirectX code is typically highly verbose, with even simple projects requiring hundreds to thousands of lines in C++ code. For the sake of brevity, this thesis will only provide superficial examination of the key functions required for raytracing.

Initialization of a DXR raytracer typically follow these common steps [3]:

- Initializing the DirectX device (GPU) and verifying that it supports raytracing.

- Loading scene geometry and generating a BVH acceleration structure from it.

- Loading and compiling the respective HLSL shaders, defining root signatures and shader tables.

- Defining a DirectX pipeline state object.

- Dispatching a workload to the pipeline.

3.4.5 Acceleration Structure

As described in section 2.7.3, utilizing a hierarchical acceleration structure can reduce the complexity per ray from linear to logarithmic in the number of triangles. There are a variety of acceleration structure types available and DirectX does not mandate the use of any particular one, though bounding volume hierarchies (BVHs) are generally best suited [3].

The construction process and data structure of DirectX BVHs is entirely opaque, as they are built and maintained by the device driver on the GPU [3]. Different graphics card vendors may choose alternative structures, but DirectX operates on a given set of structural principles [3, 55]:

The acceleration structure consists of two levels: a bottom-level acceleration structure (BLAS), which contains geometric primitives, and a top-level acceleration structure (TLAS), that contains one or more bottom-level structures (see fig. 3.10).

---

[3]Some functionality given in the example, such as the functions CameraPosition(), isTransparent() and PixelPosition() are not part of DXR and have been abstracted for the sake of simplicity.
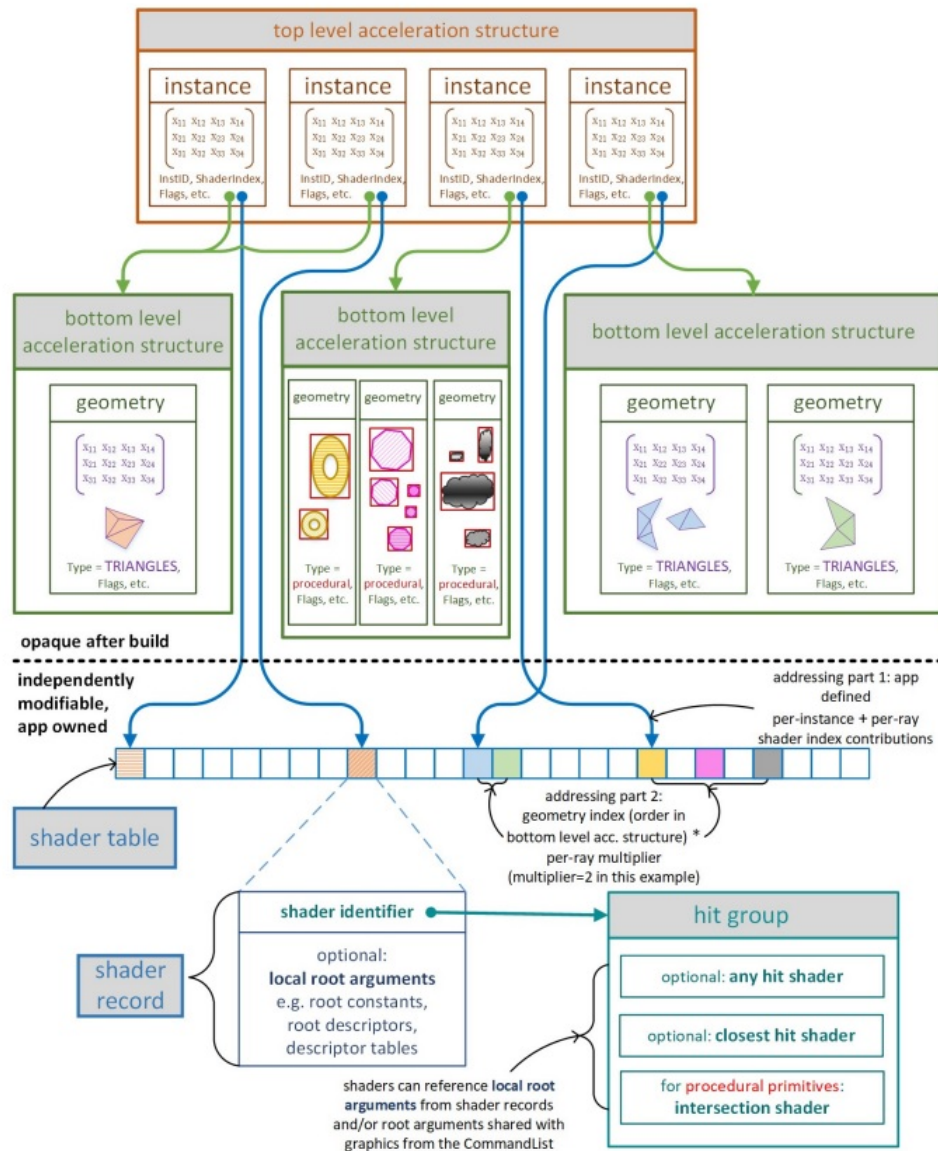
FIGURE 3.10: Key components of an RTX pipeline: TLAS, BLAS and shader binding table. Image from the Microsoft DXR specification [55].

BLAS

A bottom-level acceleration structure is usually a BVH in of itself that represents a single geometry type. Ray-triangle intersection tests are performed on BLAS data [3].

    If the geometry topology remains fixed, BLAS structures can be refit with scene changes, which is an order of magnitude faster than complete rebuilds. However, repeatedly performing refit operations may degrade the quality and performance of the acceleration structure over time. It is generally recommended to use an appropriate combination of refits and complete rebuilds [3].

TLAS

Analogously to how single 3D models can be instantiated multiple times with individual model matrices, BLAS instances within a TLAS are referenced with memory pointers alongside a transformation matrix [55] (see fig. 3.11). Whilst the reuse of geometry has great benefits to memory requirements, its overuse can impact performance, as the individual BLAS instances ought to overlap as little as possible [3].

A TLAS is, in essence, an acceleration structure of acceleration structures. Pointers to BLAS structures already living in GPU memory are contained alongside an instance matrix as well as other data like shader-index, flags etc.
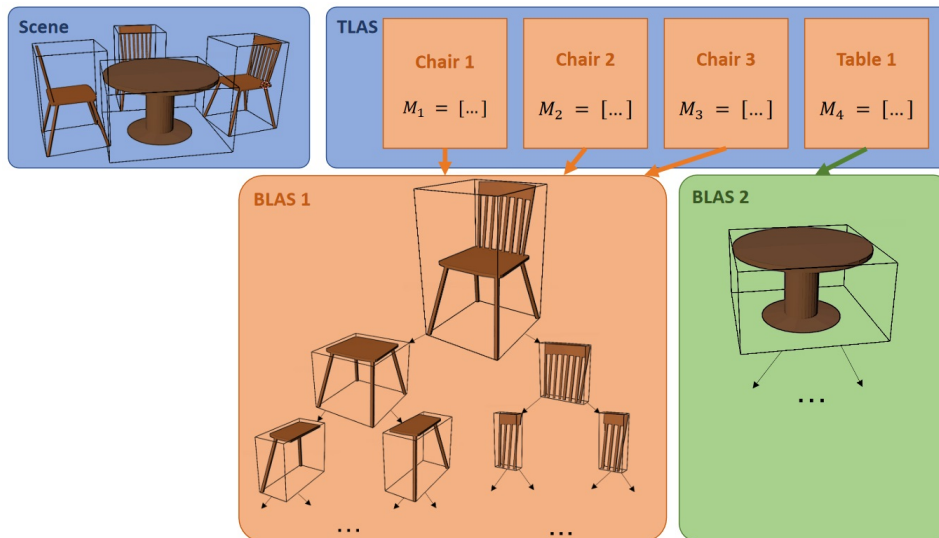

FIGURE 3.11: Simplified illustration of how the individual components of a scene relate to BLAS-TLAS components.

Shader Table

The whens and hows of tracing rays in DXR are not as strictly mandated by a sequential pipeline as in rasterization. As such, all resource bindings and shaders must be simultaneously available during the entire execution time. The selection of which shader to run is treated as any other resource binding and kept as a set of shader records in a contiguous region of memory known as the *shader table*.

The shader binding table is, in essence, a region of 64-bit aligned GPU memory that is owned and managed by the application [55]. It links the acceleration structures, hit groups and shader functions together by indicating what programs are executed for which geometry and which resources are associated with it [55].

3.5 Status Quo of RTX

In their examination of RTX technology, Sanzharov et al. conclude that whilst RTX on Turing GPUs performs well, its software-emulated variant that runs without RT-cores is an inefficient and expensive process that "essentially loses to simple and straightforward open source ray tracing" [35], implying that "'the golden age of software' has ended and that 'the golden age of compilers and HW/SW projects' has started" [35].

The visual fidelity that hardware-accelerated RTX can provide has indeed found favour with developers, as implied by its widespread adoption across many products and rendering engines such as software by *Adobe, Unity* and a vast catalogue of video games [80]. Even more predicating is the fact that one of Nvidia's primary competitors on GPU market, *AMD* released their newest line of products, the *Radeon RX 6000 Series*, with an RT-core equivalent component, designed specifically for hardware-accelerated raytracing [81].

Despite these successes, lower *frames-per-second*, in addition to price and compatibility constraints, show that RTX remains an *enhancement*, not a replacement, to classical rasterization. The underlying concept of raytracing remains just as computationally expensive as it has been since its inception in the late 1970s.

In the same time, rasterization-based techniques have come a long way in finding ideal mathematical approximations, shortcuts and simplifications that maximize photorealism. Combining the newest kinds of these techniques can produce images of similar quality to raytracing, albeit at far greater speeds.

FIGURE 3.12: Screen-space reflections (left) vs ray-traced reflections (right). Screen-space reflections are generally less resource-intensive, but can only reflect geometry that is rendered to the screen itself. Images taken from the PC game *Hellblade: Senua's Sacrifice* by *Ninja Theory*.

A great example of how clever and sophisticated these techniques have become can be observed in the form of *screen-space reflections* (fig. 3.12). This is a fast method of creating realistic looking reflections by simply taking respective pixel values already rendered into the FBO [36]. At minimal performance impact, this requires no triangle-intersections or BVH traversals, but only geometry that is visible on screen can be reflected [36].

Raytracing, on the other hand, provides a more powerful, brute-force approach that costs exponentially more. Consumers may prefer incurring the small cost in photorealism provided by raytracing in return for a more responsive application running at a higher frame-rate.

It is generally recommended by Nvidia themselves, to opt for a hybrid approach by using rasterization as a base and complement it with raytracing where it provides the most benefits (such as specular reflections, refractions and shadows) [49, 11].

In chapter 2 we demonstrated that radiosity and raytracing share many fundamental aspects by deriving both from the rendering equation. The underlying implication, which we will examine in the subsequent chapters, is that the performance increase RTX provides for raytracing ought to applicable to radiosity as well.

Chapter 4

RTX Radiosity

The preceding chapters delineate how Nvidia's RTX technology functions and how it can be leveraged through the DirectX 12 API.

In this chapter we present a concrete implementation in the form of an RTX-based radiosity application referred to as *RTRad*. We commence by first exhibiting the primary components of regular, progressive radiosity, which we later expand to incorporate *refinement* (and more) in chapter 5.

4.1 Status Quo of GPU-based Radiosity

Radiosity variants or derivations are used industry-wide in many real-time rendering engines. Yet despite their immense potential for paralellization, many radiosity implementations continue to perform better on high core-count CPUs as opposed to GPUs.

In their measurements, Carr et al. found that although CPUs currently perform better in matrix-based radiosity calculations, a GPU's performance scaling is significantly closer to linear, albeit with a fairly constricting upper limit, due to a GPU's limited memory capacity [16].

The algorithm finds itself in an unusual predicament, where it is neither particularly well suited for CPU nor GPU execution. Each radiosity patch can be processed in parallel on a GPU, but solving the visibility function V requires a data structure to be traversed *sequentially*, a process better suited to the powerful cores of a CPU.

Some solutions, like the one proposed by D'Azevedo et al. [37], attempt to tackle this imbalance by dividing the workload onto a hybrid GPU/CPU platform, where only the view-factors are computed GPU-side via a compute-shader, then read back into CPU memory and finally used in a rudimentary CPU-based solution.

The introduction of RT cores strikes a compromising balance between the few, high-performance cores of a CPU and the many, low-performance cores of a GPU. Offloading the visibility component of radiosity onto this new hardware may prove to be an ideal solution to this bottleneck.

4.1.1 Lead-up to RTRad

Raytracing and radiosity are both based on the rendering equation and, as such, share many fundamental mathematical components. The primary goal behind *RTRad* is to demonstrate that the performance increase RTX provides to raytracing is applicable to radiosity as well.

In this chapter we set out to alleviate the visibility bottleneck by substituting techniques based on hemicuboid z-buffering with an RTX-based solution, which should prove more adequate for for random rays requiring random memory-access [35].

The introduction of the Turing architecture opens the door to an entirely new set of specialized computation units on GPUs that may well be useful in areas beyond their intended use-case. Similarly to how regular graphics processors, initially intended purely for 3D rendering, have found themselves beneficial for purposes such as cryptography and machine learning, *RT cores* may also prove themselves useful for applications outside of the real-time raytracing domain.

With the implementation of an RTX-based radiosity algorithm, we put forth the general argument that RT cores can offer a great computational shortcut for highly accurate visibility simulations in general (see fig. 4.1).
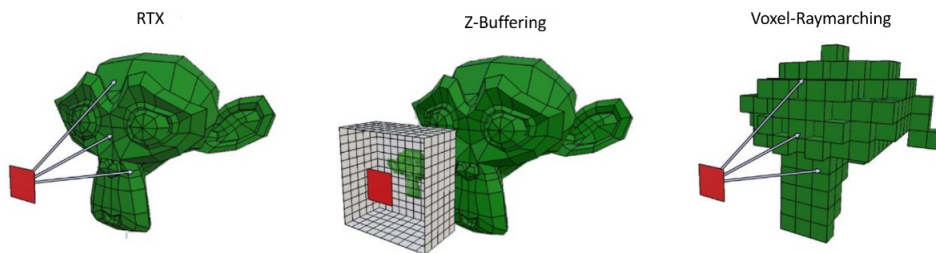

FIGURE 4.1: Different methods for calculating visibility. This thesis examines how RTX and voxel-raymarching hold up against z-Buffering.

4.1.2 Target Use-Case

The implementation presented in this thesis is fully capable of quickly producing high-quality radiosity lightmaps that can be used as textures for diffuse global illumination. However, the underlying software is primarily intended as a research project and proof-of-concept, not a consumer-targeted application to be used in production-ready applications.

The underlying use case an RTX-based lightmapper could cover in practice, is mainly oriented towards developers or designers of 3D environments working on specialized workstations with RTX-compatible graphics cards. The faster computation time would facilitate a more comfortable workflow and, once computed, the lightmaps can be mapped onto geometry and displayed to end-users on any PC at virtually no cost at all.

4.2 Previous Work

The relative novelty of RTX means its application or examination in anything other than real-time raytracing is quite sparse.

According to its documentation, contemporary versions of *Unreal Engine* make use of RTX in their *GPU Lightmass* system [56], although it is not clear in what manner or capacity. Shcherbakov et al. hint at the idea of potentially utilizing RTX in the future to accelerate their *Dynamic Radiosity* algorithm [38], and Lin advocates for its usage to accelerate the VPL generation process in instant radiosity [82].

Radiosity implementations running on GPU hardware *without* RTX have existed for some time alongside several noteworthy publications describing them.

4.2.1 GPU Radiosity

As part of the 2005 *GPU Gems 2* book [5], Coombe et al. provide an early implementation of the progressive refinement radiosity algorithm that runs on common GPUs using z-Buffering for visibility.

Alongside a series of other acceleration techniques, this solution can ultimately "compute a radiosity solution of a 10,000-element version of the Cornell Box scene to 90 percent convergence at about 2 frames per second" [5].

This implementation has served as a primary influence on this thesis with several core concepts, such as using GPU-generated mipmaps to decrease shooting resolution, being derived from it.

4.2.2 Rapid-Radiosity (RRad)

Not to be confused with the program presented in this thesis (RTRad), *RRad* [61] is a GPU-based implementation that served as a direct lead-up to this thesis, made with the open-source API OpenGL. Completed as a software project part of the computer graphics lecture at the Freie Universität Berlin, it highlights with clarity how visibility is the only major hurdle that prevents the widespread adoption of GPU-based radiosity.

RRad approximates a scene through basic geometric shapes (spheres and triangles) and then loops over each shape and performs a simple, discrete ray-intersection on each. This geometric approximation is hard-coded into the shader's code itself, making the application simple and lightweight, but entirely unsuitable to complex environments. Fig. 4.2 shows the default RRad scene with a lightmap of 512×512 pixels, for which a single bounce of light requires approx. 2 seconds of computation time on a GeForce RTX 2070S GPU.
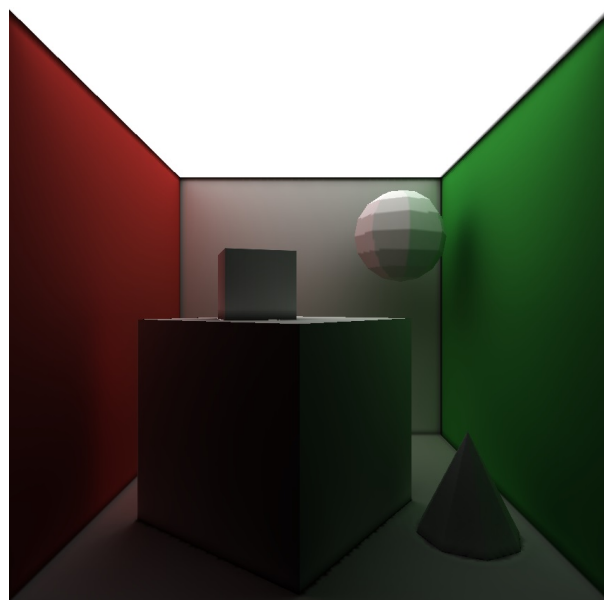

FIGURE 4.2: Simple render done with RRad. The GPU implementation means that performance is decent, but the amount and complexity of the shapes (spheres and triangles) is highly limited, which are all tested for intersections sequentially.

Many fundamental design choices presented in this thesis have their roots in RRad, with the most substantial change being the replacement of the discrete, hard-coded raytraces that run on CUDA cores, to a fully flexible RTX solution.

4.3   Source Code and Dependencies

RTRad was developed in C++ 17 with Nvidia's own *Falcor* as the underlying framework. Visual Studio 2022 served as the primary IDE and the program was exclusively tested on a system with an Nvidia RTX 2070S GPU and an AMD Ryzen 3900X CPU.

The complete source code, a demonstration video, as well all executable files for the finished project can be found on the following github repository[1]: `https://github.com/Helliaca/RTRad`

4.3.1   Falcor

Falcor is an open-source framework intended specifically for rapid prototyping of real-time rendering applications [62, 83]. Maintained as well as internally utilized by Nvidia, it provides a considerable set of advanced graphics features such as stereo rendering for VR, physically based shading and, most importantly for the context of this thesis, built-in RTX support [83].

Additionally, there are thin abstraction layers on top of DirectX 12 that reduce the amount of redundant, verbose DirectX code required for a functioning application as well as convenient UI and profiling systems.

RTRad is built on Falcor 4.4, leaving some advanced features available in later versions (Faclor 5.2 being the most recent at the time of writing) such as DLSS aside.
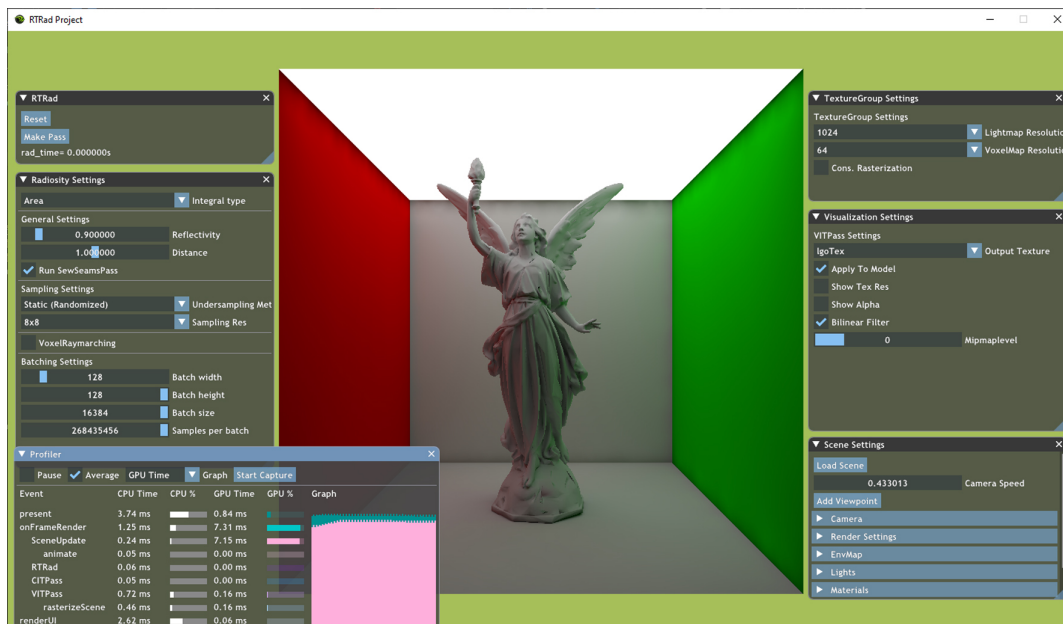

FIGURE 4.3: GUI of the RTRad application. Each render pass has its dedicated GUI window to adjust settings, alongside scene controls, pipeline controls and a profiler.

---

[1]The latest commit ID at the time of writing is `056e0e1b7cc89190231a6fbb1e81bd04ac6e0701`.

4.4 Program Structure

The employed programming patterns were kept consistent with the precedent set in Falcor's source code and respective educational content (see [62, 83, 84]), including the usage of factory methods to create objects and referencing them with smart pointers to avoid memory leaks.

The core program takes the form of a graphics-pipeline that consists of several graphics passes which are executed and managed by a central manager object based on Falcor's *IRenderer* interface.

4.4.1 Class Structure

RTRad follows the class structure depicted in fig 4.4: A *BasePipelineElement* class serves as a base class that provides a custom GUI method, so that each pipeline component can manage their own GUI elements that are used to adjust settings which are bound to their respective parents through the *SettingsObject* template (see fig. 4.3).
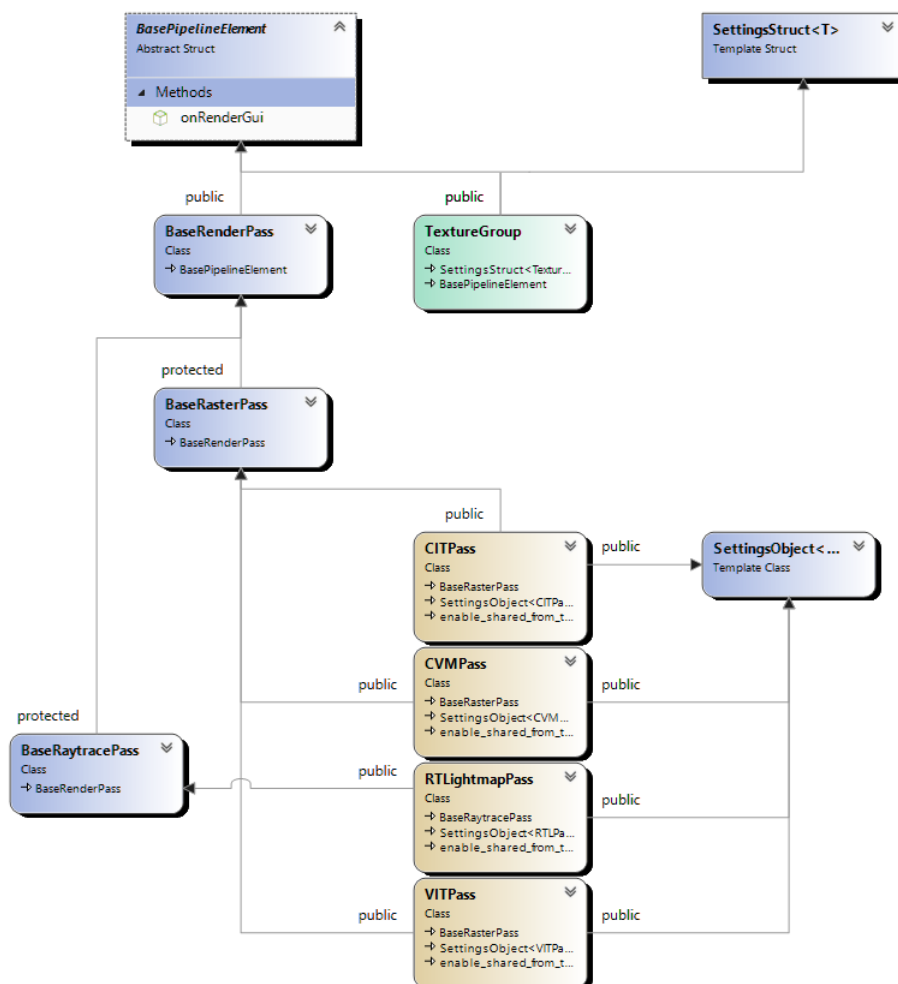

FIGURE 4.4: Simplified class diagram of RTRad. The TextureGroup class (green) represents the input/output data container, which is processed by up to four subsequent passes (orange).

A *BaseRenderPass* serves as the base class for render passes which form the backbone of the pipeline. A central *RTRad* object manages its execution and data-flow. Each pass self-manages its UI elements, shaders and shader-uniform variables. A *TextureGroup* serves as the input-output data structure which is passed through the pipeline.

4.5 Input Data

The underlying goal is to take a scene, consisting of geometry and materials, and generate a bitmap texture containing diffuse global illumination as the output, where each pixel corresponds to a radiosity patch.

Mapping each pixel in the output texture to a surface in 3D space is accomplished through the scene's *UV coordinates*.

4.5.1 Subdivision through UV Mapping

The process of ascribing each 3D vertex an additional 2D coordinate on a texture is called *UV mapping*. Utilizing this process for radiosity patch placement comes with the benefit of streamlining the data-containers for each input. Increasing or decreasing the resolution of the lightmap can occur seamlessly, as every patch points to a texture-coordinate correspondent to it.

There exist a plethora of algorithms to automatically generate UV unwrappings for any 3D model or scene [39]. In practice, rendering engines tend to come equipped with their own built-in tools specifically meant for automatic lightmap UV generation [57, 58].

Making these unwrappings as seamless and efficient as possible is a unique challenge beyond the scope of this thesis. All unwrappings utilized in RTRad were created manually, or with the tools contained in the *Blender* 3D modelling software.
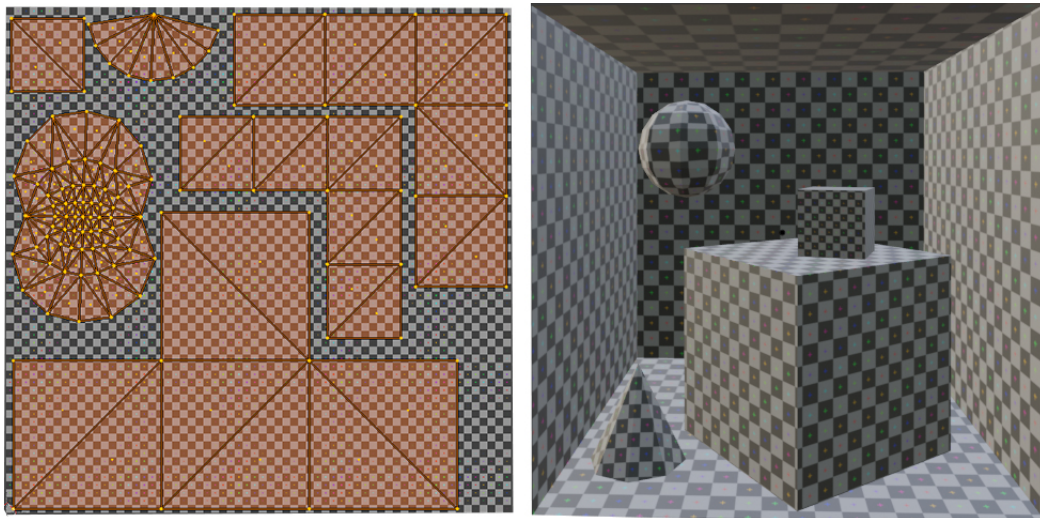

FIGURE 4.5: The default RRad scene [61] (right) and its UV coordinates (left) with a checker pattern applied as a texture. Each square of the checker pattern would correspond to a radiosity patch.

4.5.2 Input Components

Recall the Monte-Carlo approximation for view factors established in (2.37):

$$F_{i,j} = \frac{1}{K} \sum_{k=1}^{K} A_j \frac{\cos \theta_{ki} \cos \theta_{kj}}{\|x_{ki} - x_{kj}\|^2} V(x_{ki}, x_{kj}) \tag{4.1}$$

$\cos \theta_{ki}$ can be derived as the dot product between the normalized normal vector of the sample point and the normalized vector pointing from x_{ki} to x_{kj}. As such, the exact data required to calculate a form factor $F_{i,j}$ is the following:

- Surface area A_j

- Normal vectors n_{ki} and n_{kj}

- World-space positions x_{ki} and x_{ki}

- Visibilities of the the two locations $V(x_{ki}, x_{kj})$

Since the processing order of rays is not deterministic, it is imperative that all of these data points for all patches are available at all times of a radiosity iteration. We accomplish this by pre-computing individual textures that contain each data-point for all patches. The conglomeration of these textures is aptly named *TextureGroup*, and serves as the primary input-output data structure for the entire application.

Texture-Group

Each individual texture contains different information, but conforms to the exact same UV mapping. If, for instance, we require the normal vector of a patch, we simply perform a look-up operation on the texture containing normal vectors.

In total, following textures make up the texture group:

- *pos*: World-position

- *nrm*: World-space normal vector (normalized)

- *mat*: Material properties (color)

- *arf*: Surface area of each patch

- *lig_{in}*: Input lighting - i.e. emission - values of the current iteration

- *lig_{out}*: Output lighting texture to write to

Pixels that are not occupied by any geometry at all, are marked as 'non-patches' by setting their alpha value in the *pos* texture to zero. When sampling for lighting contribution, only pixels with a non-zero alpha value are processed.
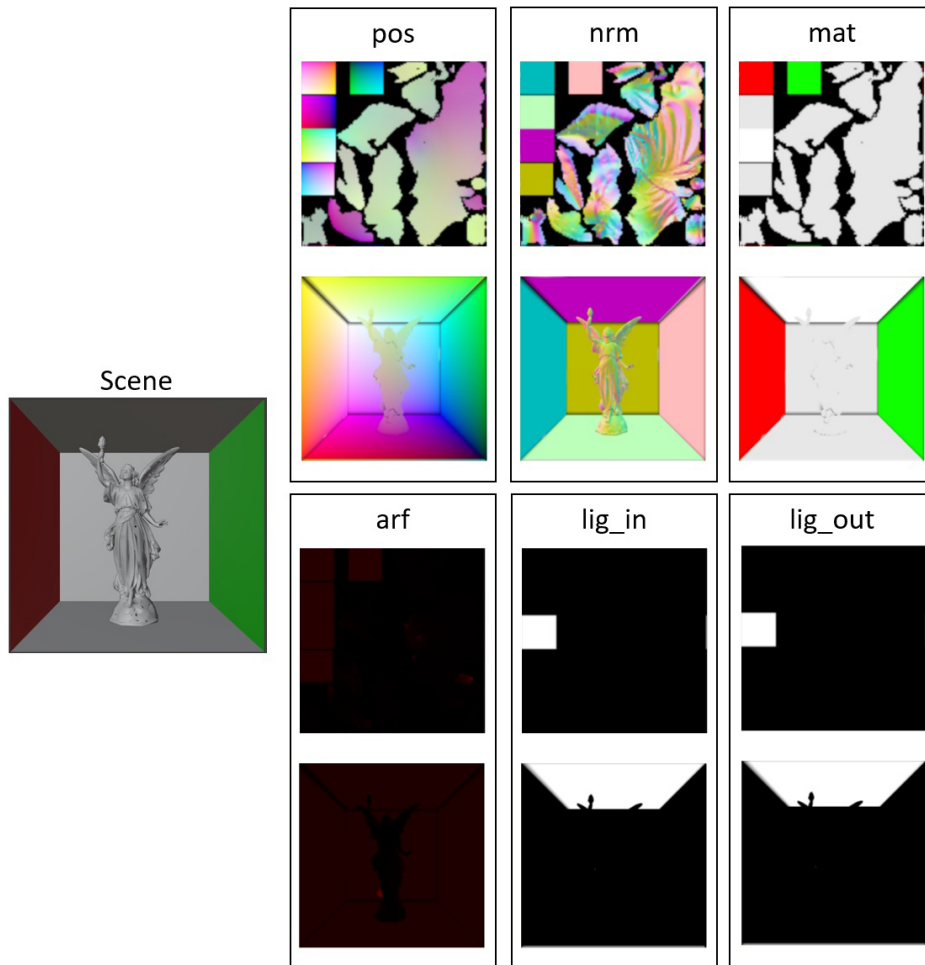
FIGURE 4.6: Original scene (left) and corresponding texture group at a resolution of 128×128 (right). The bottom rows corresponds to each texture being mapped back onto the geometry.

4.6 CITPass

The *create input textures pass* is a separate, a-priori rasterization pass that generates the above described textures from the scene's 3D geometry.

It involves applying a custom vertex shader that places each vertex into a position in clip-space that corresponds to its UV coordinates. Whilst a typical vertex shader commonly applies an objects model, view and projection matrices like so:

$$vert(v) = P * V * M * v.pos \tag{4.2}$$

our custom vertex shader for the CITPass simply applies a vertex's UV coordinates:

$$vert(v) = \begin{bmatrix} 2 * (v.uv.x - \frac{1}{2}) \\ 2 * (v.uv.y - \frac{1}{2}) \\ 0 \\ 1 \end{bmatrix} \tag{4.3}$$

UV coordinates range from zero to one, whilst clip-space is defined as the $[-1,1]$ range, hence the subtraction of $\frac{1}{2}$ and multiplication by 2. We assume the UV mapping to contain no overlapping geometry, making the z-coordinate produced by this vertex shader irrelevant.

With the vertex shader in place, setting the output resolution to the desired resolution of the input textures then ensures that the subsequent pixel shader is executed exactly once for each pixel - e.g. each patch - in these textures.

An example how how this texturegroup looks like after the CITPass can be seen in fig. 4.6. The lighting textures lig_{in} and lig_{out} store the emissive values L_e for each patch, which serves as the radiant-exitance for the first radiosity pass.

4.6.1 Surface Area

Whilst normal vectors, positions and material properties are easily passed into the pixel shader through the rasterization pipeline, surface areas of patches require additional information that a simple, barycentric interpolation cannot provide.
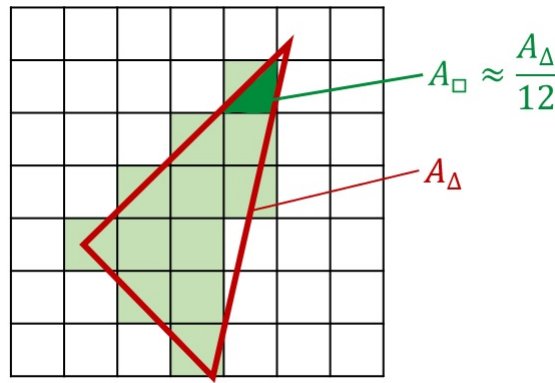

FIGURE 4.7: Approximation of a patches surface area: Since this triangle occupies 12 patches on the lightmap, the world-space surface area of each patch can be approximated by ascribing each patch the surface area of the triangle divided by the amount of patches.

Our textures consist of pixels, effectively squares, which correspond to radiosity patches. The set of pixels rastered by a triangle, will only *partially* cover or be covered by it. As such it is difficult to calculate an accurate value for the world-space surface area each patch possesses, for which we employ the following approximation: *The surface area of any one patch is equal to the surface area of the underlying triangle divided by the total amount of patches occupied by said triangle.*

Let $A_\triangle^w$ be the world-space surface area of the triangle, with $A_\triangle^{uv}$ being its surface area on the $[0,1]$ bounded UV map. The amount of patches a triangle occupies can be derived from the product of its UV surface area and the total amount of patches in the texture n. For instance, if a triangle occupies half of the UV map, its UV surface area will be equal to $\frac{1}{2}$, which implies that it is represented by $\frac{n}{2}$ patches.

The world-space surface area $A_\square^w$ of a single patch can then be computed from the relationship between this product and its world-space triangle $A_\triangle^w$ (see fig. 4.7):

$$A_\square^w \approx \frac{A_\triangle^w}{A_\triangle^{uv} * n} \tag{4.4}$$
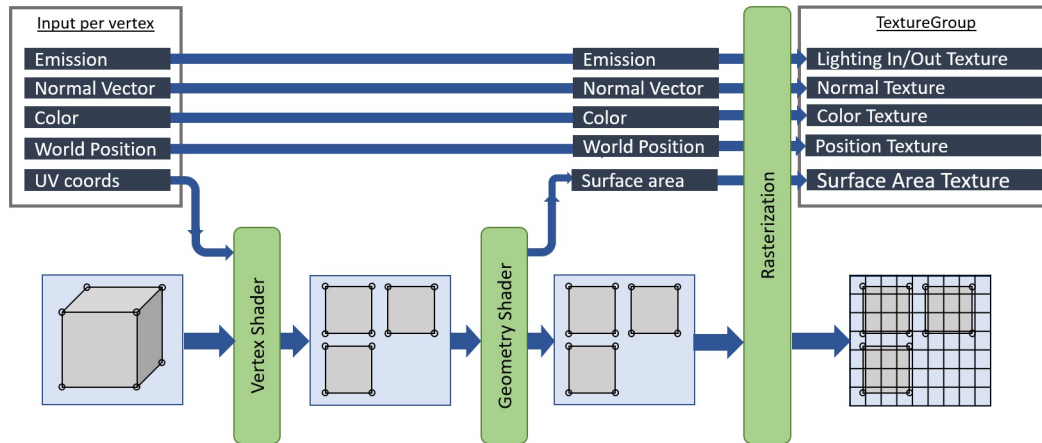
FIGURE 4.8: Dataflow diagram of the CITPass, resulting in the texturegroup that serves as the input-output datastructure for subsequent passes.

We perform this operation on a per-triangle basis within the geometry shader of the CITPass. Unlike other common applications of geometry shaders (see 3.3.1) we do not modify the output vertices, but merely append the fraction above as an attribute to each vertex, which then gets passed on to the pixel shader that stores it in the surface area texture *arf*.

The overall execution and information flow of the CITPass is visualized in fig. 4.8.

4.7 RTLPass

The central component of our application is the *ray-traced lightmap pass* (RTLPass), which takes the generated texturegroup as its input values and produces an output texture correspondent to the lightmap after a single iteration of progressive radiosity. For each subsequent pass the output lighting-texture is swapped and fed back into the algorithm as the input for the next iteration (see fig. 4.9).

The RTLPass is the only DXR pass (non-rasterization pass) in the entire application. It launches the ray-generation shader for each pixel in the lighting texture, which commences a separate GPU thread for each radiosity patch which loops over all the other patches to sum up their respective lighting contribution into the output texture[2].

The overarching procedure of this pass closely resembles the progressive radiosity algorithm given in section 2.8.4 and can be summarized with the following pseudo-code:

---

[2]Note that since visibility is symmetrical under $V(x_1, x_2) = V(x_2, x_1)$, only half the rays given in algorithm 3 are theoretically required. Unfortunately, in practice this leads to multi-threading memory collisions that are described in section 4.7.4.
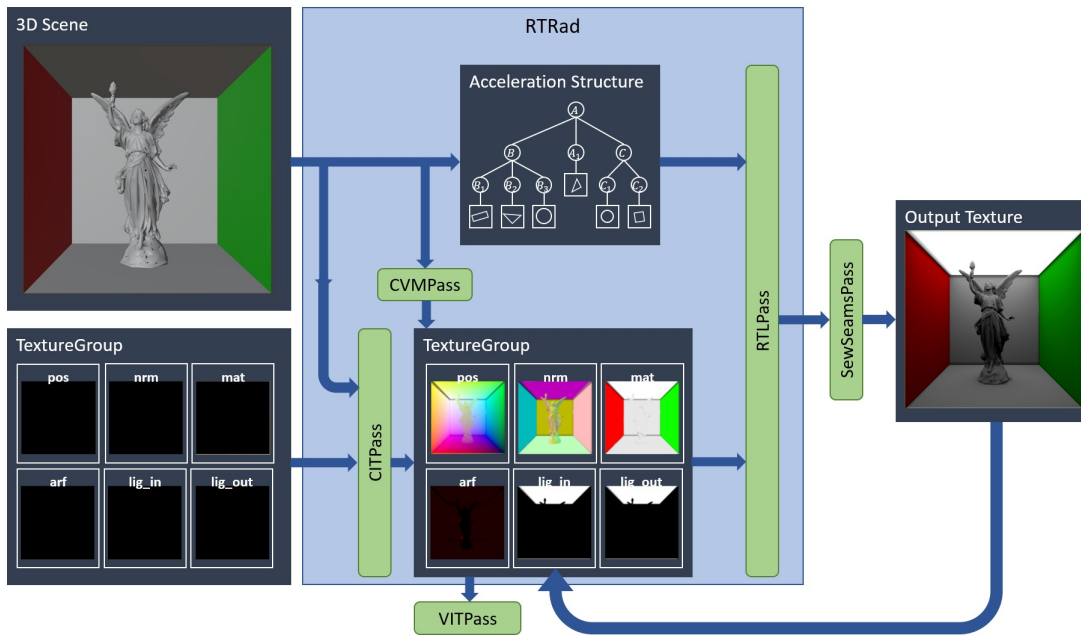
FIGURE 4.9: Generalized overview of information flow in RTRad. The 3D scene is used by the CITPass to populate a blank texturegroup which, alongside the acceleration structure, serves as the input for the RTLPass. The output texture of the RTLPass can then be used as input for the next iteration. Textures can be visualized on screen through a *VITPass* (see 4.9). The purpose of *CVMPass* and *SewSeamsPass* are described in sections 5.3 and 4.8 respectively.

Algorithm 3 RTLPass

1: **for** $i \in [0, n]$ **do** ▷ For each patch (executed in parallel)
2:     $L_{out}(i) \leftarrow L_e(i)$ ▷ Set initial lighting value
3:     **for** $j \in [0, n]$ **do**
4:         **if** $j \neq i$ **then** ▷ For every other patch
5:             Shoot a ray from $pos(i)$ to $pos(j)$
6:             **if** no geometry is encountered along the way **then**
7:                 Calculate view factor $F(i, j)$
8:                 $L_{out}(i) \leftarrow L_{out}(i) + mat(i) * F(i, j) * L_{in}(j)$ ▷ Add contribution
9:             **end if**
10:         **end if**
11:     **end for**
12: **end for**

4.7.1 Visibility Raytracing

As defined in section 2.9.2, two locations x_1 and x_2 are mutually visible if a ray launched from one towards the other, arrives at the other unimpeded:

$$V(x_1, x_2) \leftrightarrow I(x_1, x_2 - x_1) = x_2 \tag{4.5}$$

The insight gained from the shader execution order pictured in chapter 3 (fig. 3.9) implies that, in order to minimize the required amount of BLAS traversal the

ray lengths ought to be kept as short as possible in addition to stopping execution upon any intersections.

As an alternative to the equation above, two points x_i and x_j are also mutually visible if a ray from x_i towards x_j with some offset ε and of length $|x_i - x_j| - 2\varepsilon$ does not encounter any geometry at all, which in RTX would trigger the execution of the miss shader.

Shortening the rays to a length of $|x_i - x_j| - 2\varepsilon$ and then determining visibility through the *miss shader* requires fewer intersection tests, binding-table lookups and should generally speed up the algorithm, as any non-visible pairs can be quickly discarded through the `RAY_FLAG_ACCEPT_FIRST_HIT_AND_END_SEARCH` flag.

In our implementation, we use the ray payload to get indices on the origin and destination patches into the miss shader. Alternatively, one could also store a boolean that represents visibility and then perform lighting calculations in the ray-generation shader itself.

Below we provide a simplified, superficial version of our RTLPass shader code:

```
// Custom ray-payload
struct RayPayload
{
    uint2 origin_coord;       // sampler patch
    uint2 destination_coord; // sampled patch
};

[shader("raygeneration")]
void launchRays()
{
    uint2 origin_coord = DispatchRaysIndex().xy;

    for (uint x = 0; x < lightmap.width(); x++) {
        for (uint y = 0; y < lightmap.height(); y++) {

            uint2 destin_coord = uint2(x, y);

            float3 origin_pos = pos[origin_coord];
            float3 destin_pos = pos[destin_coord];

            RayDesc ray;
            ray.Origin = origin_pos;
            ray.Direction = destin_pos - origin_pos;
            ray.TMin = EPSILON;
            ray.TMax = distance(origin_pos, destin_pos) - 2 *
                EPSILON;

            RayPayload rpl = { origin_coord, destin_coord };

            TraceRay(scene,
                RAY_FLAG_ACCEPT_FIRST_HIT_AND_END_SEARCH,
                0xFF, 0, 0, 0
                ray,
                rpl
            );
```

```
35            }
36        }
37  }
38
39  [shader("miss")]
40  void primaryMiss(inout RayPayload rpl)
41  {
42        // Lighting calculations.
43        // Determine what light rpl.destin_coord contributes onto
               rpl.origin_coord
44  }
```

4.7.2   View Factor Calculation

From the results of some of our early prototypes we deduced that increasing the amount of samples in a Monte-Carlo based view factor was operationally equivalent with simply increasing the resolution of the underlying lightmap itself. Instead of sampling a single patch multiple times, this would essentially divide the patch up into several smaller ones (see fig. 4.10).
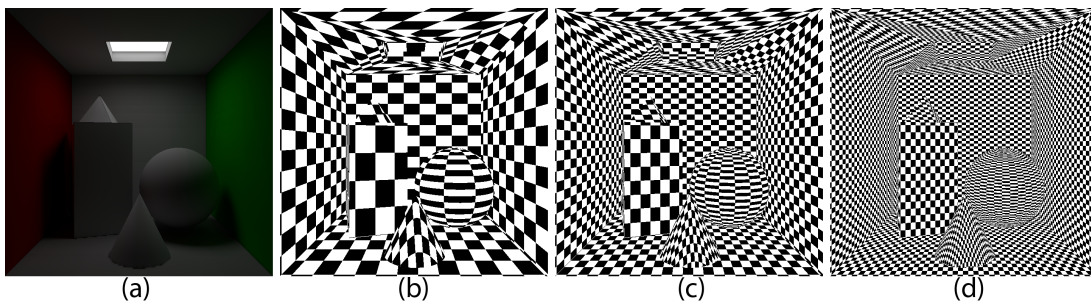

(a)   (b)   (c)   (d)

FIGURE 4.10: Original scene (a) and patches for a lightmap of 32×32, 64×64 and 128×128 pixels respectively (b-d). Each black or white rectangle corresponds to a single patch.

In light of the foregoing, we determined that a Monte-Carlo view factor with $K = 1$ was sufficient, where larger values of K can effectively be simulated by larger lightmaps as well as our employed Monte-Carlo *undersampling* (see section 5.1.1). As such, we are able to compute view factors in a single step through the following formula:

$$F_{i,j} = \frac{1}{K} \sum_{k=1}^{K} A_j \frac{\cos \theta_{ki} \cos \theta_{kj}}{\|x_{ki} - x_{kj}\|^2} V(x_{ki}, x_{kj}) \approx A_j \frac{\cos \theta_i \cos \theta_j}{\|x_i - x_j\|^2} V(x_i, x_j) \qquad (4.6)$$

Bearing in mind the individual textures given in section 4.5.2, allows us formulate this equation as the exact programmatic steps our shader undertakes to compute the view factors between two patches *i* and *j*:

$$F(i,j) = arf(j) \frac{(nrm(i) \cdot \frac{pos(j)-pos(i)}{\|pos(j)-pos(i)\|}) * (nrm(j) \cdot \frac{pos(i)-pos(j)}{\|pos(i)-pos(j)\|})}{\|pos(i) - pos(j)\|^2} \qquad (4.7)$$

where $nrm(i)$ represents a texture-lookup on the normal-vector texture for patch i etc.

4.7.3 Lighting Contribution

Combining the factors from the radiosity equation given in 2.34 with the view factor definition given above, yields the total lightflow from a patch j to another patch i as the following:

$$L(j \rightarrow i) = lig_{in}(j) * mat(i) * \frac{\rho}{\pi} * F(i,j) \tag{4.8}$$

where ρ is the reflectivity constant, mat is the color of the material and lig_{in} is the input lighting texture (equivalent to the emission texture on the first pass).

This equation can be seen reflected in the code of our miss shader. A simplified version of said shader is listed below:

```
void AddContribution(uint2 self_c, uint2 other_c) {
    // World positions
    float3 self_wpos = pos[self_c].xyz + minPos;
    float3 other_wpos = pos[other_c].xyz + minPos;

    // Distance
    float3 self_to_other = other_wpos - self_wpos;
    float r = length(self_to_other) * distance_factor;

    // Cosines
    self_to_other = normalize(self_to_other);
    float3 self_nrm = nrm[self_c].xyz;
    float3 other_nrm = nrm[other_c].xyz;
    float self_cos = dot(self_nrm, self_to_other);
    float other_cos = dot(other_nrm, -self_to_other);

    if (self_cos <= 0.0f || other_cos <= 0.0f) return;

    // Form factor
    float F = arf[other_c].r * self_cos * other_cos * (1.0f /
        (PI * r * r));

    // Apply contribution
    lig_out[self_c] += lig_in[other_c] * mat[self_c] *
        reflectivity_factor * F;
}
```

We found that several custom tweaks not shown in the pseudo-code above, such as clamping lighting contribution and form factors to certain maximum values, significantly improved lighting quality.

4.7.4 Indexing and Memory Conflicts

In theory, the reciprocity rule (2.36) implies that a view factor between two patches only needs to be computed once, with its inverse resulting from a simple multiplication in the form of $F_{ji} = \frac{A_i}{A_j} F_{ij}$. This simple fact has far-reaching implications in that dramatically fewer texture-lookup and ray-trace operations are required, effectively cutting the required computational expense in half. Unfortunately, it does not play out this trivially in practice.

A naive implementation may look as follows:

Algorithm 4 RTLPass - Reciprocity Rule

1: **for** $i \in [0, n]$ **do**
2:      $L_{out}(i) \leftarrow (0, 0, 0)$
3:      **for** $j \in [i + 1, n]$ **do**
4:          Shoot a ray from $pos(i)$ to $pos(j)$
5:          **if** no geometry is encountered along the way **then**
6:              Calculate view factor $F(i, j)$
7:              $L_{out}(i) \leftarrow L_{out}(i) + F(i, j) * L_{in}(j)$
8:              $L_{out}(j) \leftarrow L_{out}(j) + F(i, j) * \frac{arf(i)}{arf(j)} * L_{in}(i)$
9:          **end if**
10:      **end for**
11: **end for**

Note in particular line 3, where the inner for-loop commences at $i + 1$. Under this offset the complexity of the algorithm is lowered to its Gaussian sum $\frac{n^2 + n}{2}$ which, despite ultimately boiling down to $O(n^2)$, still provides a considerable gain in performance.

The crux of this approach emerges from parallelization, as multiple threads modifying a single patch cause *memory write collisions*. Specifically, parallelizing the outer for loop in algorithm 4 will cause such collisions on line 8, as multiple separate threads adopt the same value for j. If the inner loop is parallelized, the same issue will occur on line 7.

Thread safety features, such as mutex locks, come with their own respective performance overheads and are not available in common GPUs. If ignored, these write conflicts manifest themselves as unusual artifacts depicted in fig. 4.11. The resulting smudges can be allayed under specific conditions using certain batching parameters, though the results are highly unreliable[3].

For a more reliable solution, following options are available:

- Adhering to our original algorithm 3, where each thread is assigned a singular patch i that it can write to. No writing conflicts will occur, albeit view factors and rays will have to be computed twice for each patch-pair.

- These duplicate calculations can be prevented by temporarily caching their results. For instance, a thread processing patch i would temporarily store the visibility and/or view factor towards another patch j in memory, so that when the thread processing patch j samples patch i, the cached value can simply be retrieved. Unfortunately, GPUs have notoriously limited memory capacities

---

[3]The results in fig. 4.11 have been produced with a lightmap resolution of 128×128 and a batch size of 64×64. On our hardware, different batch sizes sway the rate and intensity of these imperfections, but finding the ideal size largely comes down to trial-and-error.
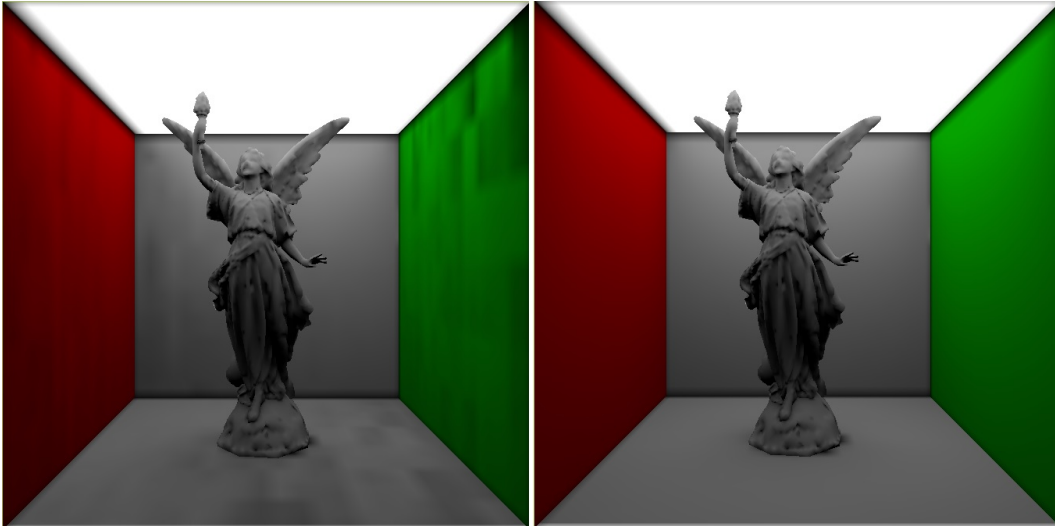
FIGURE 4.11: Results when applying a regular algorithm 3 (right) and when applying the flawed algorithm 4 (left). Note in particular the grey smudges on the walls that result from collisions of multiple threads writing simultaneously to the same memory address. The location and intensity of the smudges are mostly consistent across multiple runs, but differ depending on input parameters.

and the amount of memory required lies in $O(n^2)$. Our testing concluded that caching visibility data *can* be viable for small lightmaps. A detailed account on our findings regarding visibility caching can be found in section 5.2.

- Since Turing GPUs contain between 40 to 80 RT cores, a rough limit of 40-80 threads can be assumed to be writing to memory simultaneously. An indexing solution ensuring that any patch-pairs which may potentially write into shared memory are far enough apart in the index-sequence may diminish the effects of memory-conflicts sufficiently to make the final result identical.

Most multi-threaded, progressive refinement solutions implement the first solution by calculating all view factors on-the-fly. Naturally, the performance balance depends on whether the benefits of GPU parallelization can outweigh the cost incurred by doubling the amount of visibility calculations.

Our implementation runs, by default, with the same approach but also allows visibility caching to be enabled for smaller lightmaps (see section 5.2).

4.7.5 Batching

Operating systems typically expect applications to remain responsive in scheduling. Forcing an indefinitely large workload onto the GPU will cause the program to be terminated by the OS after exceeding a certain time threshold, which can be reached rather quickly by a workload as complex as progressive radiosity.

To circumvent this problem, we compute lightmaps in a series of batches, as opposed to all at once. The batching process is managed by internally the RTLPass object, which allows the user to adjust certain batching parameters through its GUI. These parameters allow defining the dimensions of the rectangle - or strip - that is computed with each batch.

By gradually testing various thresholds, we determined an upper limit of 128^4 rays traced per batch to be adequate for our specific hardware. Depending on the sampling and resolution settings, the RTLPass object ensures that batching parameters are dynamically adjusted in order to remain at or below this limit.

We recommend maintaining the batch size as close to this limit as possible, because a larger batch count (such as a single patch per batch) inevitably comes with a large overhead cost in setting up each individual render pass.

4.8   SewSeams Pass

When patches are not aligned with geometry, lights and shadows can leak and produce undesirable protuberances in the form of unrealistic shadows or highlights [69].

Increasing lightmap size or adjusting UV coordinates can eliminate these issues entirely, but is difficult to do automatically. Some re-meshing solutions generate edges along predicted discontinuities of the radiosity function [69], though this only functions if each patch is given its own primitive.

Similar inaccuracies (labeled as *geometry leaks* in fig. 4.12) can result from pixels neighbouring a patch that does not have its center covered by a surface, as these will not be rasterized by the CITPass. These types of leaks can be far more frequent than the former and will not only manifest themselves in corners.

Conservative rasterization prevents this issue, but produces additional problems, as it overwrites patches when occupied by several primitives in addition to annulling our approximation for surface area (see section 4.6.1).
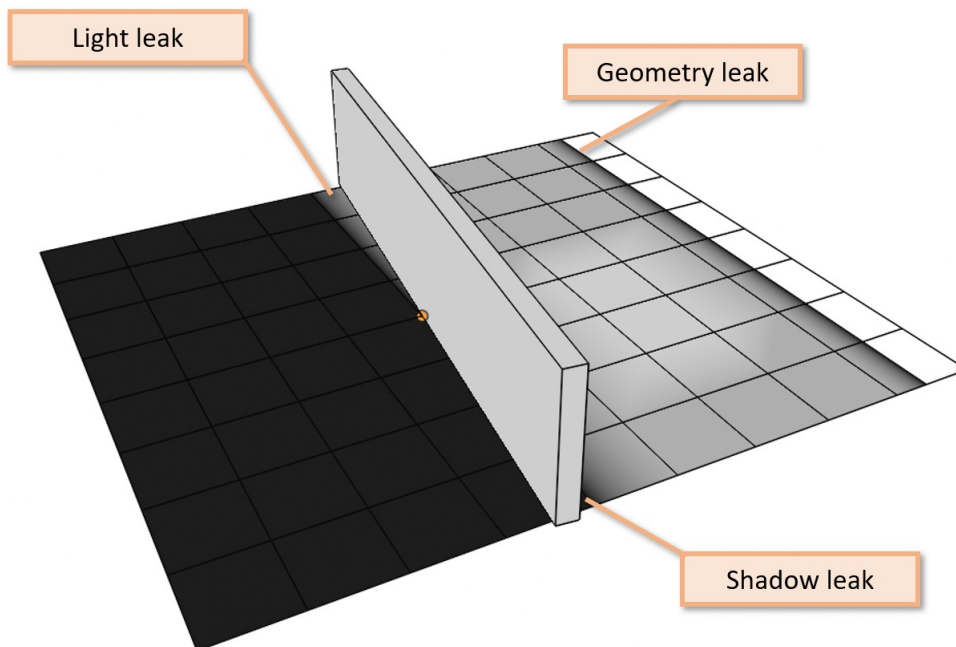

FIGURE 4.12: Light and shadow leaks on UV-mapped lighting textures. Geometry leaks occur when a patch/pixel is only marginally covered by a surface.

We employed a custom solution in the form of an additional *SewSeams* pass that is executed after each radiosity iteration.

This pass runs for every lightmap pixel that was not rastered by the CITPass and thus is not treated as a patch. If any of the pixel's neighbours *is* a patch, then the pixel assumes its color. This process effectively expands the UV mapping of each cohesive shape by a margin of one pixel, which eliminates the vast majority of geometry-based leaks entirely (as shown in fig. 4.13).
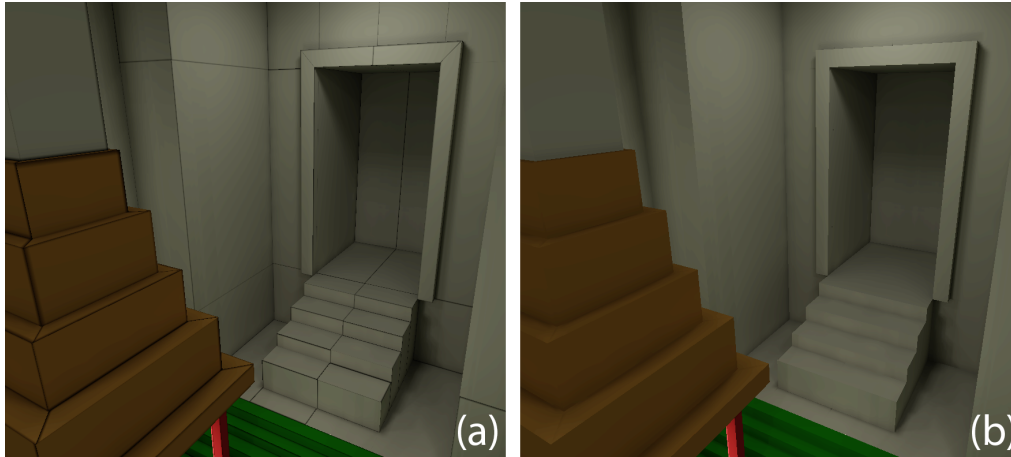

FIGURE 4.13: Scene without (a) and with (b) a *SewSeams* pass.

4.9 VITPass

For visualization purposes we employ an additional *visualize input textures pass* (VIT-Pass), which serves as a tool to display and analyze input and output data. Whilst this pass may not be an intrinsic part of our algorithm, it certainly proved invaluable to debug, optimize and analyze our implementation.

We employ a plethora of settings that can be accessed through Falcor's UI to adjust precisely what output should be displayed and in what form. The user can choose which mipmap level of which texture to display and whether to render them as texture, masked texture or applied to the underlying 3D model.

As is common practice in radiosity, we employ a *bi-linear* magnification filter to smooth out each texture, which can be toggled on or off (see fig. 4.14).

The VITPass is built on a custom vertex and pixel shader, which allow for a wide variety of different setups. Additional information such as lightmap resolution, refinement-nodes or voxel-maps can also be adequately visualized (see fig. 4.15).

All scene renders or raw textures contained in this thesis are produced using the VITPass (unless stated otherwise).
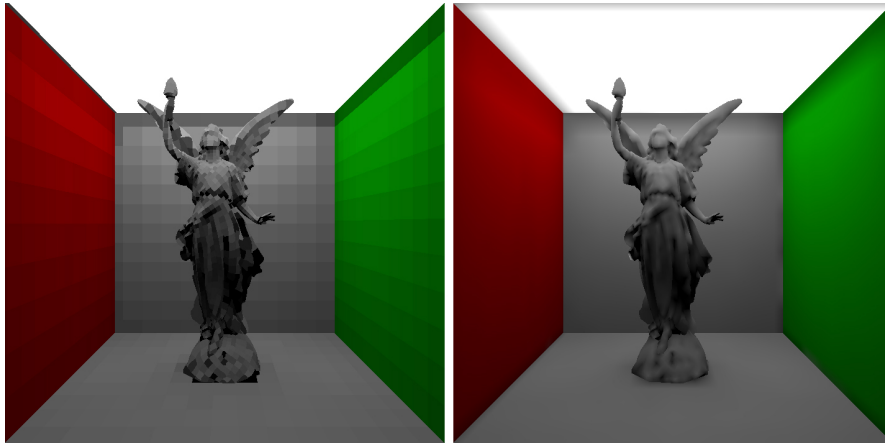
FIGURE 4.14: Low resolution lightmap (64 × 64) without (left) and with (right) a bilinear magnification filter.
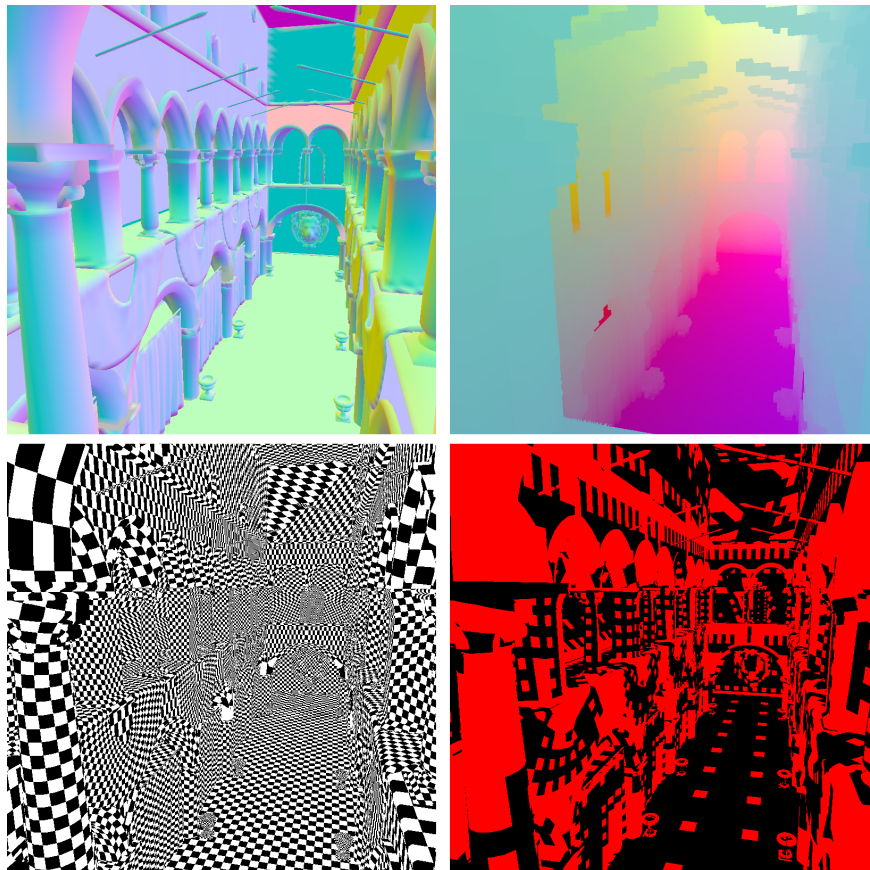

FIGURE 4.15: Examples of data visualization through the VITPass. left-to-right, top-to-bottom: Normal vectors, voxel map, texture resolution and quad-tree of a scene.

Chapter 5

Performance Improvements

Put together, the components outlined in the previous chapter already constitute a fully functional implementation of the progressive radiosity algorithm. In this chapter we expand it to include *refinement* in addition to further performance enhancements and variations commonly seen in other implementations.

5.1 Refinement

In traditional radiosity, patches are assumed to have a uniform radiant exitance across their surface [22, 29]. Since ray-traces are of logarithmic complexity [3], our doubly nested, pair-wise for-loop results in an altogether complexity of $O(n^2 \log t)$ where t is the number of primitives and n is the amount of patches. This complexity suggests that lowering the amount of required samples (e.g. the "contributing" patches n) yields a far greater positive impact on performance than diminishing the value of t.

There are several different approaches of accomplishing this goal, which were broadly categorized in section 2.8.5. In this chapter we primarily focus on methods based on *h-refinement*, namely *static undersampling* and *adaptive subdivision*. Our approaches follow the precedent set by the implementation presented in the GPU Gems 2 [5], by separating the *sampled* patches as a proper subset of the overall lightmap. Our different strategies of generating such a subset are depicted in fig. 5.1.

5.1.1 Static Undersampling

A crude and simple method of sampling at a lower resolution than that of the lightmap is to simply do so through a static stride. The nature of UV unwrapping implies that neighbouring pixels on the lightmap likely also represent neighbouring patches in world space, thus increasing the viability of discarding samples belonging to these. Whilst no common name has been established for this method, we will refer to it as *static undersampling*.

In its simplest version, we simply discard all but the upper, left-most pixel of each consecutive square consisting of m pixels (the *sampling window*).

This inherently degrades the algorithms' complexity to $O(\frac{n^2}{m} \log t)$, albeit the quality of the resulting lighting may likewise suffer in proportion to m. The decreased amount of samples needs to be reflected in the lighting contribution values (as given in 4.7.3) as a factor of m, which is the amount of patches each sample represents:

$$L(j \to i) = m * lig_{in}(j) * mat(i) * \frac{\rho}{\pi} * F(i, j) \tag{5.1}$$

Monte-Carlo Undersampling

If the edges of a scene's UV map are aligned along one of the texture's axes, then the fixed-step nature of static undersampling may lead to certain colors or surfaces being grossly over-represented in the aggregate lighting contribution. This effect can be mitigated by selecting a *randomized* pixel from each $\sqrt{m} \times \sqrt{m}$ square.

Naturally, the employed randomization ought to be seeded on the patch-index of the contributor, not the shooter.
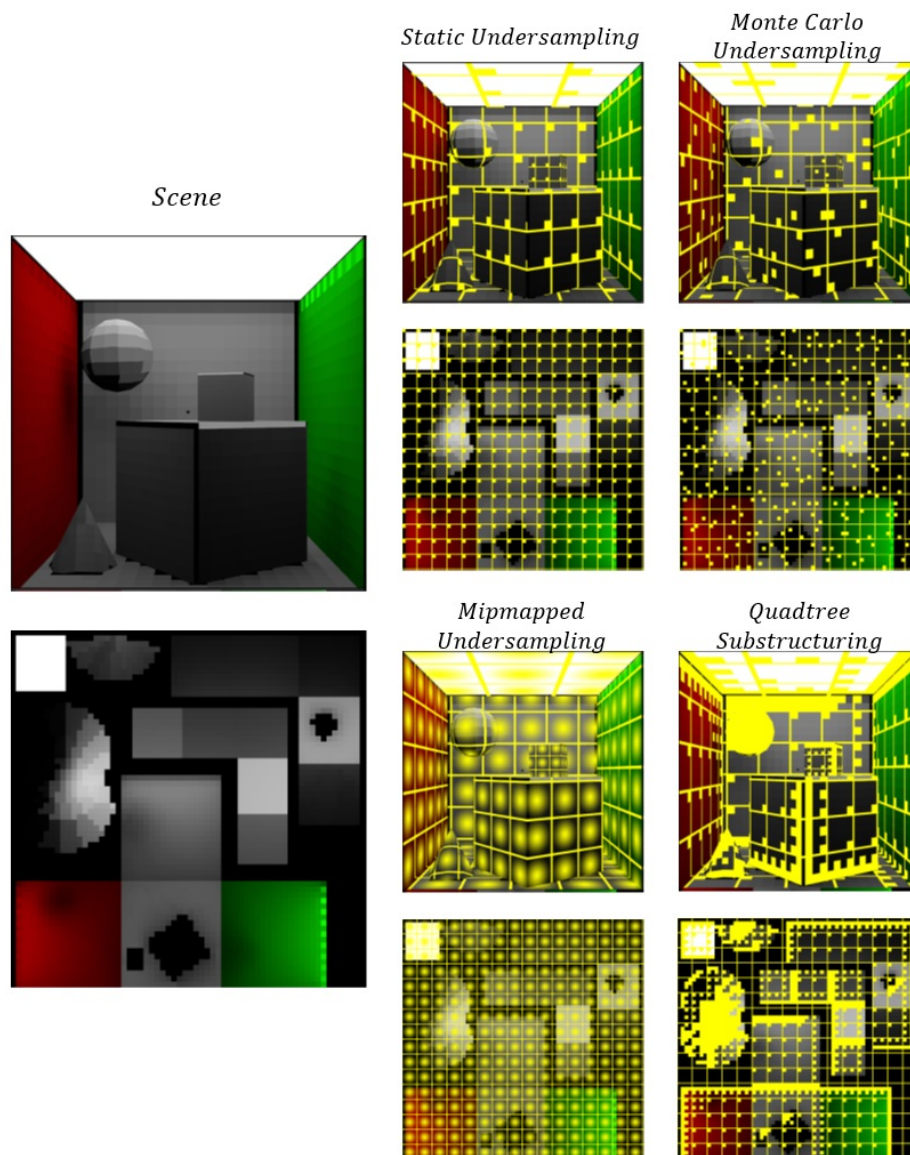

FIGURE 5.1: Different strategies for undersampling on a lightmap of 64×64 pixels. One sample is taken for each window of 4×4 pixels ($m = 16$), with the sampled patches marked in yellow. Mipmapped takes averages and thus has the best coverage, whilst substructuring (e.g. adaptive subdivision) allocates more samples to areas of a high gradient.

5.1.2 Mipmapped Undersampling

The information loss incurred by static undersampling can be placated by computing averages for each set of pixels. Some GPU implementations utilize *mipmapping* as a fast and convenient tool to accomplish this [5].

Mipmaps are a sequence of pre-calculated, down-scaled versions of a given texture. With each *level*, the image resolution is a factor of four smaller than the previous level. Their applications are primarily centered on *texture filtering* with the intent to reduce aliasing artefacts at long render distances.

The GPU employs a *minification filter* to dictate how mipmaps are derived from the original texture. The most commonly used types are the *nearest neighbour* and *bilinear* filters [8, 75]. Bilinear mipmaps are the functionally equivalent inverse of the bilinear *magnification* filter which we touched on in section 4.9: Each pixel in the minified image corresponds to the average of a 2×2 square in the level below (see fig. 5.2).

Since generating mipmaps on a GPU is a near-instantaneous process, they can be used as a tool to sample texture areas for their average color values conveniently [8, 27, 5]. Computing the average color of a $\sqrt{m} \times \sqrt{m}$ square simply corresponds to a texture lookup on the mipmap of level $\log_2(\sqrt{m})$.

Mipmapped undersampling ensures no vital patches (such as light-sources) are discarded, but can lead to colors leaking from one surface to another when sampled.

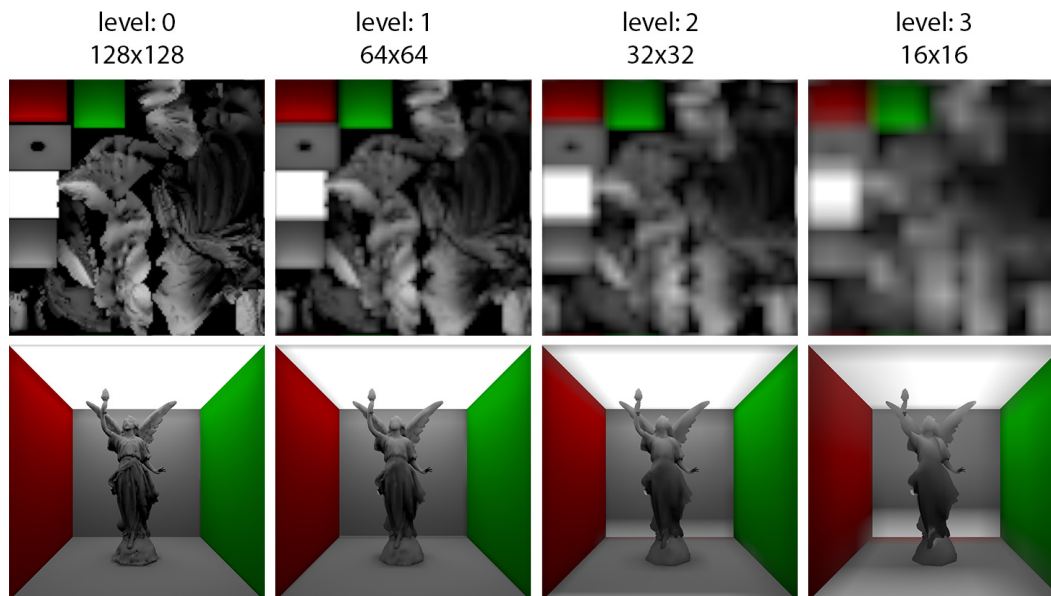| level: 0 | level: 1 | level: 2 | level: 3 |
| 128x128 | 64x64 | 32x32 | 16x16 |

FIGURE 5.2: Bilinear mipmaps of a lightmap (under a bilinear magnification filter). These are the lightmaps sampled by mipmapped undersampling with a value of $m = 4$, $m = 16$ and $m = 64$ respectively.

5.1.3 Adaptive Subdivision

Adaptive subdivision is a more sophisticated counterpart to static undersampling and is generally regarded as a core part of progressive refinement radiosity [29]. Patches that have a high gradient across them, such as shadow boundaries and penumbras, are subdivided into finer grids, whilst low detail areas are represented by a single patch [70, 29] (see fig. 5.3).

Our implementation of this concept in RTRad is loosely based on the hierarchical, quad-tree approach used by Willmott et al. [40] as well as the "progressive accuracy" solution employed by Elias [77], whilst we somewhat diverge from other GPU implementations that map each patch to a specific scene polygon and maintain a quad-tree datastructure inside a static buffer [5].

Given our emphasis on handling data directly within UV-wrapped textures, we opted to store meta-information on our quad-tree structure within the alpha-channel of the lightmap texture itself. This approach allows us to keep the algorithm shown in the previous chapter mostly intact, whilst letting us sample at a significantly lower rate.
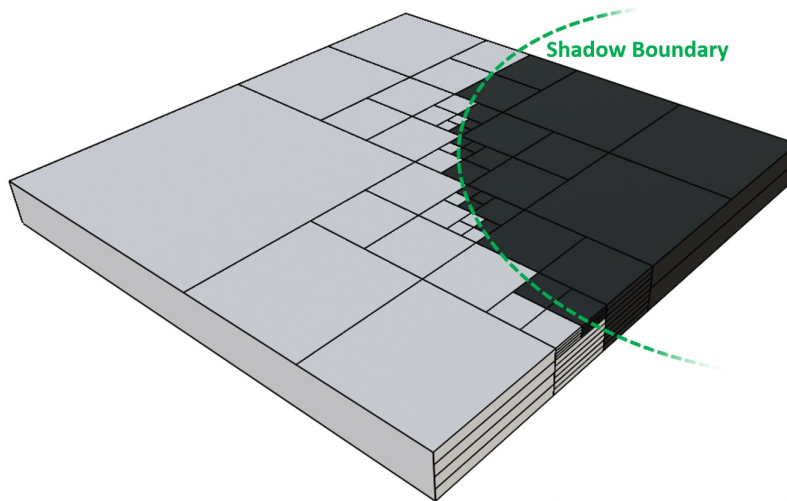

FIGURE 5.3: Quad-tree substructuring of a surface. More subdivisions are made in areas of high detail.

Alpha-Embedded Substructuring

Section 4.5.2 briefly mentions that RTRad uses the alpha channel of the position-texture to mark patches that are not occupied by any geometry and thus have no respective surface in the scene. Our quad-tree employs a similar scheme in that the alpha value of each pixel equates to the amount of patches its color value represents in sampling.

Before commencing a radiosity iteration, we run a fullscreen pixel shader on the input lighting texture that constructs the appropriate quad-tree inside the alpha channel. The tree is constructed bottom-up, with each pass of the shader processing a single level. For a maximum node size of 16×16 pixels, the required passes would thus be $\log_2(16) = 4$, equivalent to the height of a quad-tree that can represent 16×16 values under a single root.

Each pass examines the gradient of every four neighbours and determines whether these are to be merged into a single node or not. If the colors hardly deviate, the upper left-most node is designated to represent the entire group, with its siblings being dropped from the sampling pool.
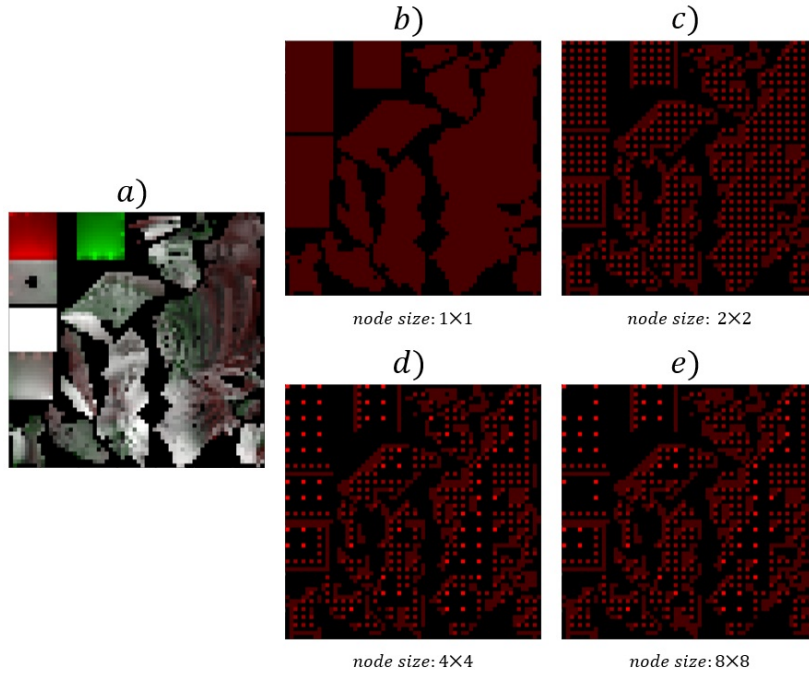
FIGURE 5.4: A 64×64 RGB lightmap (a) and its alpha channel (b-e) after 0, 1, 2 and 3 substructuring passes respectively with a gradient threshold of 0.2.

The algorithm functions in accordance to the following pseudo-code:

Algorithm 5 Alpha - Substructuring

1: **for** each pixel $(x, y) \in lightmap$ **do**   ▷ Initialize all alpha values as 1
2:     $alpha(x, y) \leftarrow 1$
3: **end for**
4: **for** $step \in \{2, 4, 8...\}$ **do**
5:     **for** each pixel $(x, y) \in lightmap$ **do**
6:         **if** x and y are divisible by $step$ **then**   ▷ For every $step \times step$ pixels
7:             $child_1 \leftarrow (x, y)$ ▷ Get 4 neighbouring child-nodes
8:             $child_2 \leftarrow (x + \frac{step}{2}, y)$
9:             $child_3 \leftarrow (x, y + \frac{step}{2})$
10:            $child_4 \leftarrow (x + \frac{step}{2}, y + \frac{step}{2})$
11:                **if** all child-nodes have $alpha(child_x) \geq (\frac{step}{2})^2$ **then**   ▷ Only merge
     nodes that have no children of their own
12:                    $g \leftarrow gradient(child_1, child_2, child_3, child_4)$ ▷ Calculate gradient
13:                    **if** $g < threshold$ **then**   ▷ Merge children
14:                        $alpha(child_1) \leftarrow \sum_{i=1}^{4} alpha(child_i)$
15:                        $alpha(child_2) \leftarrow 0$
16:                        $alpha(child_4) \leftarrow 0$
17:                        $alpha(child_5) \leftarrow 0$
18:                    **end if**
19:                **end if**
20:            **end if**
21:        **end for**
22: **end for**

As shown in fig 5.4, all occupied geometry will commence with an alpha value of one, then each 2×2 square with a low gradient is merged by transferring the alpha value of all pixels inside the square into the its upper, left-most member. The same process is repeated for 4×4 and 8×8 squares respectively.

Once the alpha-embedded quad-tree is constructed for the lig_{in} texture, the RTL-Pass will loop over all other patches, but disregard those with alpha values of zero and multiply the lighting contribution of the rest by their alpha values. The following algorithm amends our original algorithm 3 from the previous chapter:

Algorithm 6 RTLPass with Adaptive Subdivision

1: **for** $i \in [0, n]$ **do**
2: $\quad L_{out}(i) \leftarrow (0,0,0)$
3: $\quad$ **for** $j \in [0, n]$ **do**
4: $\quad\quad$ **if** $j \neq i$ and $alpha(j) > 0$ **then**
5: $\quad\quad\quad$ Shoot a ray from $pos(i)$ to $pos(j)$
6: $\quad\quad\quad$ **if** no geometry is encountered along the way **then**
7: $\quad\quad\quad\quad$ Calculate view factor $F(i, j)$
8: $\quad\quad\quad\quad L_{out}(i) \leftarrow L_{out}(i) + alpha(j) * mat(i) * F(i, j) * L_{in}(j)$
9: $\quad\quad\quad$ **end if**
10: $\quad\quad$ **end if**
11: $\quad$ **end for**
12: **end for**

The alpha channel of a standard GPU texture can typically only hold 8 bits of information which, under our algorithm, limits the maximum size of a node to $\sqrt{2^8}^2 = 16 \times 16$ pixels, amounting to a maximum quad-tree height of four. Although this limitation is negligible, as one could simply store the logarithmic or fractional alpha value instead, the remainder of this thesis will work with an upper node limit of 16×16.

Gradient Calculation

Determining an approximate gradient for a lighting texture can be accomplished in a number of ways and can incorporate data such as color values to variations in normal vectors. We based our function largely on Willmott et al. [40], which uses the standard deviation of each patch's radiosity value. We additionally include deviation in normal vectors, to ensure the edges of cohesive surfaces are weighed more heavily in the gradient:

$$
\begin{aligned}
grad(a, b, c, d) = &\frac{1}{2} \|lig(a) - mean(lig(a), lig(b), lig(c), lig(d))\| \\
&+ \frac{1}{2} \|nrm(a) - mean(nrm(a), nrm(b), nrm(c), nrm(d))\|
\end{aligned}
\tag{5.2}
$$

where, in our case, a is the upper left-most child or pixel.

Adjusting the gradient threshold respectively leads to more or fewer samples taken (see fig. 5.5). For small lightmaps, we found a value of 0.05 to provide a decent balance between performance and accuracy. Given that each consecutive RTLPass iteration has a lesser effect on lighting than the previous, a reasonable approach would be to ramp this value up successively with each pass.
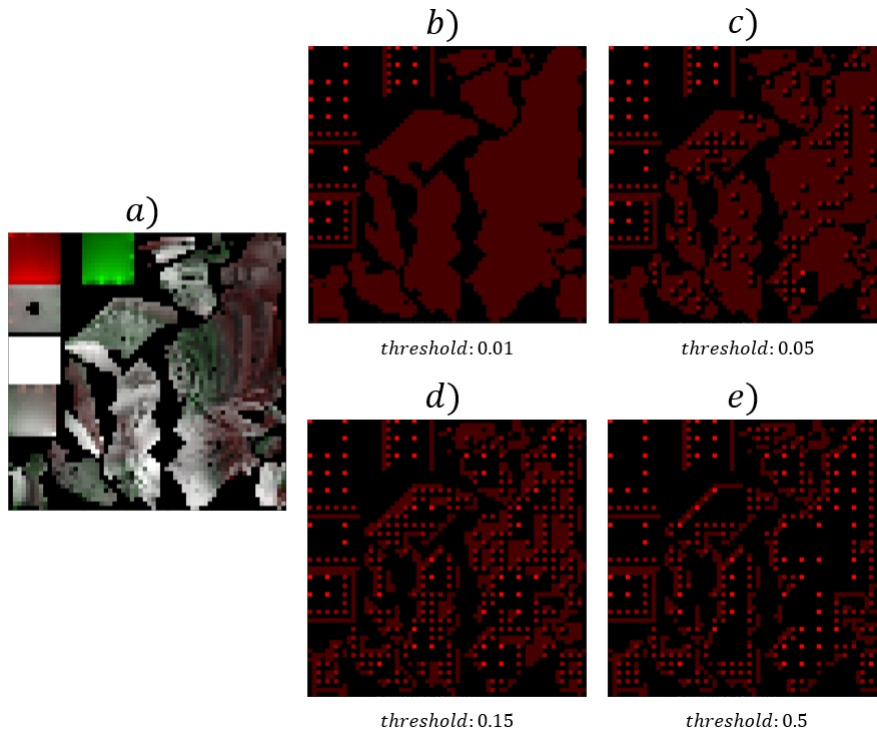
FIGURE 5.5: A 64×64 RGB lightmap (a) and its alpha-embedded quad-tree for different gradient thresholds (b-e) (black pixels are not sampled). A larger threshold will lead to more aggressive merging.

5.2 Visibility Caching

In this section we describe a visibility caching method used to complement the RTRad algorithm. The resulting performance shows promise for smaller lightmaps and is analyzed in greater detail further ahead (see section 6.4.1).

5.2.1 Memory Complexity

The required memory for all pairs of n radiosity patches lies in $O(n^2)$. In a naive implementation, a modest lightmap of 256x256 pixels would thus require 256^4 bits (approx. 0.5GB) of storage, which already borders a critical threshold of what lower grade GPUs can accommodate.

Fortunately, not every pair of patches needs to be considered. Self-referencing pairs of the form $(0,0), (1,1)...(n,n)$ as well as mirrored pairs $((x,y)$ or $(y,x))$ can be discarded, bringing the required raw memory down to $\frac{1}{2} * n^2 - n$ bits. Built-in compression algorithms are unlikely to alleviate this problem, as their intended use is centered on compression of geometry, textures or z-Buffers [41].

Our implementation stores visibility information as an uncompressed, contiguous buffer of bits which we append to our existing texturegroup. We chose an upper limit of 2^{32} bytes, as common GPUs tend not to have more than 8GB of onboard memory, and Falcor disallows the allocation of larger buffers. Theoretically, a buffer of this size can provide full coverage of lightmaps with sizes up to 512×512.

Figuring out which memory address (index) each patch pair is assigned in the visibility buffer, requires a bijective mapping function between unique pairs and respective indeces.

5.2.2 Cantor Pairing Function

Pairing functions uniquely encode two natural numbers into a single one. There are a wide variety of them each with their own use cases and respective advantages [73, 42]. For visibility caching we encode each unique pair of patches to an index in a cohesive memory sequence, which makes the *Cantor pairing function* a good choice, as it traverses a 2D grid in a triangular shape which can be modified to exactly cover each unique pair and no more (see fig. 5.6).

A vanilla cantor pairing function for non-negative integers follows this formula [42]:

$$Cantor(x,y) = \frac{x^2 + 3x + 2xy + y + y^2}{2} = x + \frac{(x+y)(x+y+1)}{2} \quad (5.3)$$

We ensure pair uniqueness by sorting x and y in ascending order. We also mirror the x-axis to ensure that at a given cutoff point (namely $\frac{1}{2} * n^2 - n$) only unique pairs have been covered:

$$address(x,y) = Cantor(n_x - min(x,y), max(x,y)) \quad (5.4)$$

where n_x is the lightmap resolution along the x axis.

In this memory-sequence function, the parameters (patches) are encoded as a single number, whilst on the lightmap they are given by two coordinates. Converting between each format is a trivial process:

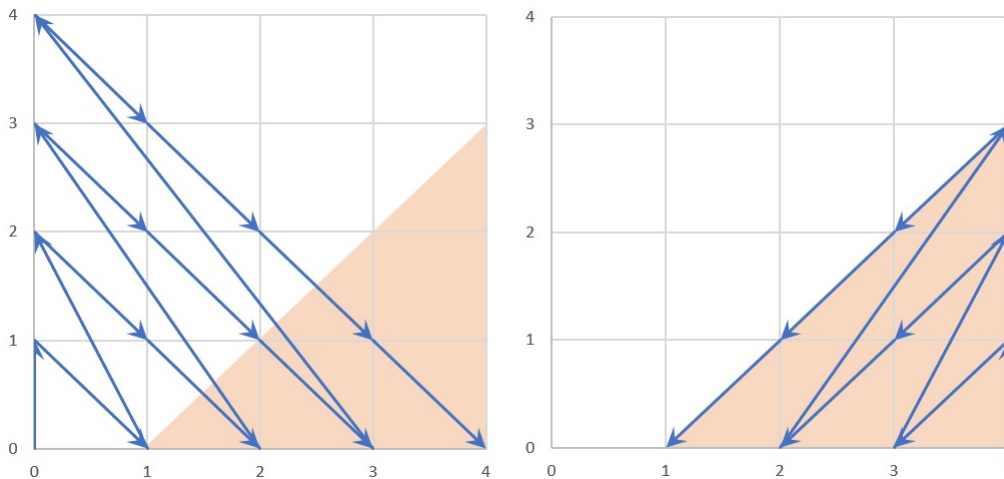$$patch_{1D} = patch_{2D}.x + patch_{2D}.y * n_x \quad (5.5)$$

FIGURE 5.6: Vanilla Cantor pairing function (left) and our custom pairing function (right) on a 5×5 grid. The area marked in orange contains all unique, non self-referencing pairs. Up to the index of 10, our pairing function excludes both mirrored and self-referencing pairs, thus encompassing exactly this area.

5.2.3 Visibility Buffer

Visibility data is stored inside a standard DirectX *RWBuffer* of unsigned integers with a maximum size of 2^{32} bytes. The modified cantor pairing function defined above

ascribes each pair of patches a corresponding index inside the buffer. If the result exceeds the maximum value of $8 * 2^{32}$, no cached data is available and the visibility will have to be computed on-the-fly through raytracing.

Accessing the individual bits of a buffer directly is not possible in HLSL, so we employ equivalent bit-shifting functions that operate on 32-bit unsigned integers instead[1].

In theory this approach may also be used to *partially* store visibility data on larger lightmaps, though based on our measurements, this only lead to performance degradation. The visibility data required by a lightmap of 1024^2 pixels would require upwards of 500 billion bits, of which the entirety of our 4GB buffer would only cover less than 7%. The cost of having to calculate the cantor-index for *all* patch pairs heavily outweighs the advantage of being able to skip the raytraces for just 7% of them.

5.3   Voxel Raymarching

The primary mechanism behind Nvidia's RTX technology is the complete parallelization of BVH traversal, triangle intersecting and shading. Given the immense number of rays required for a full radiosity iteration, we can safely assume that the limited amount of RT cores still pose a meaningful bottleneck within this system.

We devised a simple, low memory alternative for visibility calculations that can be executed on regular CUDA cores. The underlying idea consists of idle CUDA cores tapping into the unprocessed workload when all RT cores are otherwise busy.

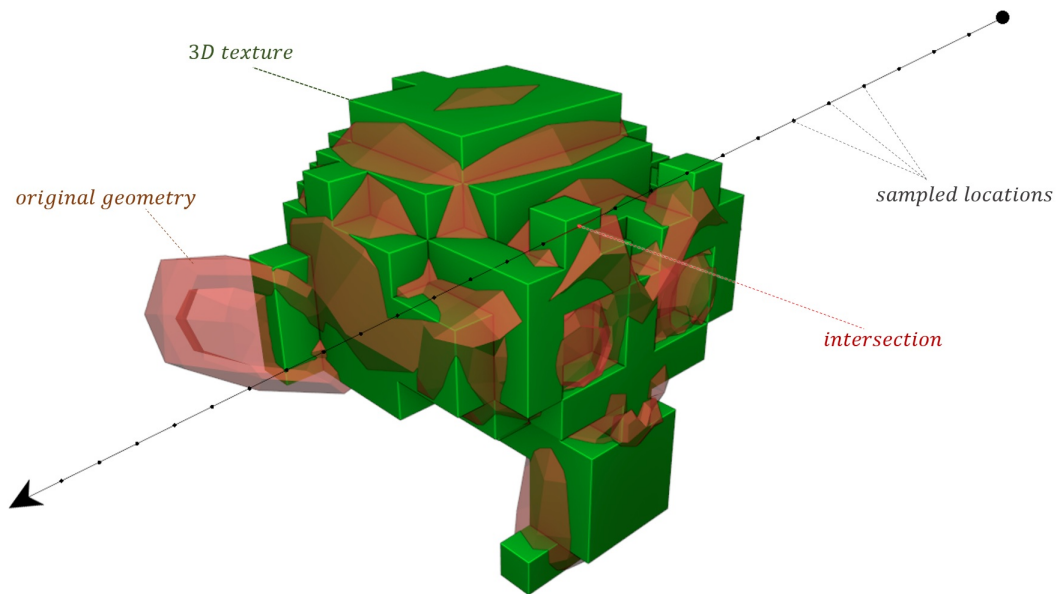The aforementioned visibility alternative has its roots in voxel cone tracing ([8, 27]).

FIGURE 5.7: Raymarching through a voxelized 3D scene. On each of the sampled locations the respective value in the 3D texture is read. In this example the ray will be marked as obstructed (i.e. not visible), because one of the sampled locations contains a value larger than zero.

---

[1]Since 32bit integers are read and written to simultaneously, this leads to similar memory collision inaccuracies as in section 4.7.4. In our testing their effects were mostly negligible for the smaller lightmaps this algorithm functions on, though became noticeable on a 512×512 resolution.

5.3.1 Raymarching and 3D Textures

Intuitively, *3D textures* function just like regular textures only with an additional depth dimension [8]. The pixel equivalents are aptly named *voxels*, resulting from a combination of the words volume and pixel.

Raymarching is a technique usually employed when surface functions are not easily solvable. Unlike raytracing, this process "marches" forward along the ray direction in a series of steps, sampling each point along the way [8]. Raymarching has a wide range of variations such as sphere tracing or SDF ray marching, which require signed distance functions [43].

Storing a voxelized representation of a scene into a 3D texture and sampling that texture whilst marching along a ray serves as a crude approximation to a regular ray-trace (see fig. 5.7).

This approach is subject to a number of limitations mostly related to its geometric inaccuracies, yet the performance remains independent of a scene's triangle count and is instead tied to the overall size of the employed 3D texture [8].

5.3.2 Scene Voxelization

To voxelize a scene into a 3D texture we employ an algorithm virtually identical to the one described by Crassin et al. [8, 27, 44], which allows an entire scene to be voxelized in a single, lightweight rasterization pass.

We run this pass as an extension to the CITPass, the *CVMPass* (*create-voxel-map-pass*), which consists of a unique vertex, geometry and pixel shader:

Vertex Shader

To ensure all surfaces are voxelized, the first step scales the scene to fit entirely into the rendered clipspace. The formula applied to each vertex simply follows from:

$$vert(v) = 2\frac{v - P_{min}}{P_{max} - P_{min}} - (1,1,1) \tag{5.6}$$

where P_{min} and P_{max} are the minimum and maximum world positions the scene encompasses. In essence, all scene vertices are linearly interpolated from a $[P_{min}, P_{max}]$ interval into the clipspace interval $[-1, 1]$.

The viewport resolution is set to be equal to the width and height of the voxelmap which, when rendered, corresponds to each primitive being projected along a texture axis [8]. The process of choosing the ideal axis of projection is performed by the geometry shader.

Geometry Shader

Rasterizing all surfaces along a single axis may leave gaps in the resulting projection if a surface's normal vector subtends a steep angle with the axis of projection [8, 71, 44]. For instance, if the green cuboid in fig. 5.8 were to be rasterized along a single axis, only one of the three surfaces would contain pixels.

In order to ensure a voxelization that covers all surfaces fully, one can either repeat the process for each axis individually or, preferably, rotate each triangle so that the dominant component of its normal vector is aligned with the axis of projection [8, 27, 85].

This process is aptly named *dominant axis selection*, and is easily performed in a geometry shader. The dominant axis of a triangle corresponds to the axis that its normal vector shares its largest component with. Commonly, rasterization applications project triangles along the z-axis, so a respective geometry shader would function like so:

Algorithm 7 Dominant Axis Selection (Adapted from VXCT [8])

1: **procedure** PROCESSTRIANGLE(v_1, v_2, v_3)
2:     $\vec{n} \leftarrow |(v_2 - v_1) \times (v_3 - v_1)|$ ▷ Calculate normal vector
3:     **for** $v \in \{v_1, v_2, v_3\}$ **do**   ▷ Rotate each vertex to maximize z
4:         **if** $n.x = \max(n.x, n.y, n.z)$ **then**
5:             $v.xyz \leftarrow v.zyx$
6:         **else if** $n.y = \max(n.x, n.y, n.z)$ **then**
7:             $v.xyz \leftarrow v.xzy$
8:         **else** $n.z = \max(n.x, n.y, n.z)$
9:             $v.xyz \leftarrow v.xyz$
10:        **end if**
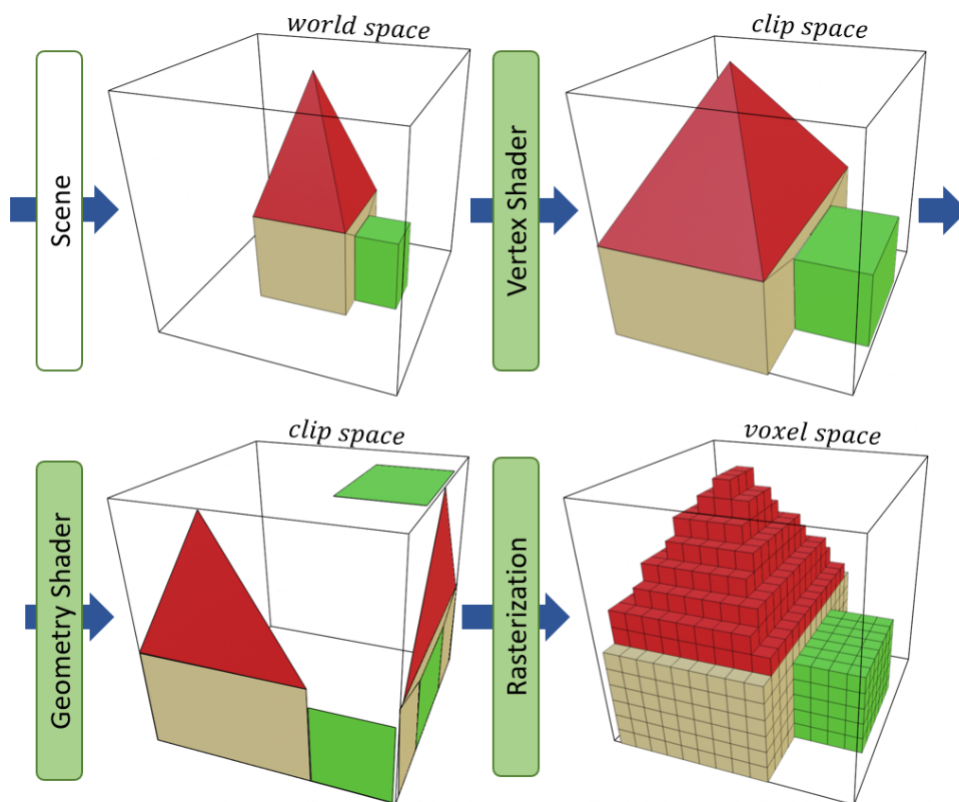11:   **end for**
12: **end procedure**

FIGURE 5.8: The individual steps of the GPU voxelization algorithm proposed by Crassin et al. Note that the geometry shader technically does not *project* the geometry, but merely rotates it so that the dominant axis is aligned with the axis of projection.

Pixel Shader

The rasterization pipeline delivers the world position that each fragment had before the geometry shader's rotations took place. For each rendered pixel that position now corresponds to a coordinate within the 3D texture that is set to be a "solid" voxel (see fig. 5.9).

In voxel cone tracing, each voxel stores information on direct lighting calculated by the phong model [8]. For our purposes it is sufficient to simply store a 0 for empty space and 1 for geometry.
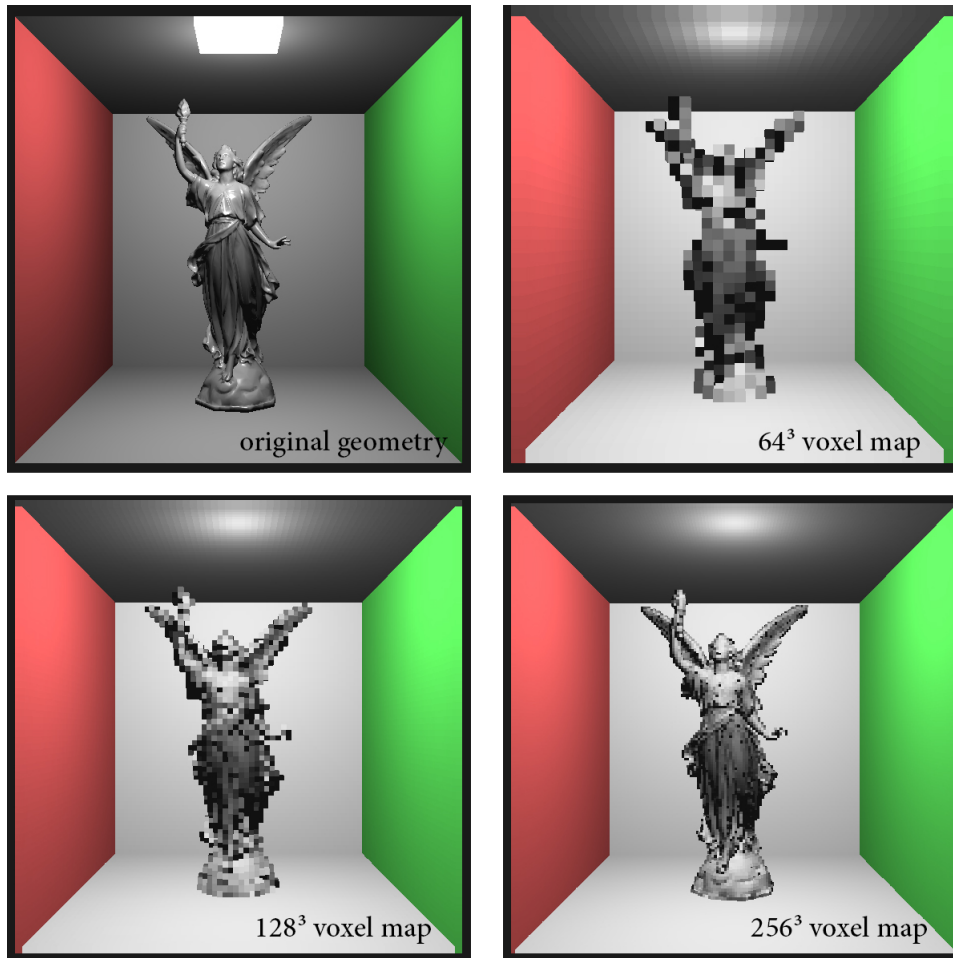

FIGURE 5.9: Phong-model direct light values stored in voxelmaps of different resolutions, as depicted by Benjamin Kahl [8].

We store the voxelmap alongside our other textures in the texturegroup. Marching through it along a ray until a voxel is solid becomes a trivial procedure that can be directly called instead of the RTX `TraceRay` function.

In RTRad, we allow the user to set a custom ratio of RTX ray-traces that are replaced by voxel-raymarches. We found that, for small voxelmaps, this could indeed lead to an improvement in pass-time but would significantly deprecate lighting quality. Nevertheless, raymarching does serve as a simple and lightweight fallback for graphics cards that do not support RTX. A more detailed account on our findings is listed in section 6.4.2.

5.4 Directional Sampling

Some radiosity implementations rely on the hemicube approximation for visibility and do not calculate view factors explicitly [5]. Instead, gathered intensity is estimated by generating samples on a hemicube and determining which patch a ray incoming from that direction would have originated on. The total gathered intensity can thus be estimated as the average for each of these directions. This process corresponds, in essence, to the same distribution that a classical raytracing program would sample for diffuse reflections. We broadly categorized this approach as "directional sampling" in section 2.8.6.

To analyze the viability and performance of RTX in these types of implementations, we included a modified version of our algorithm that relies on direction-based sampling.

For each patch, a number of rays are shot through a surrounding hemisphere. Whichever surfaces these encounter are sampled for their color and added into the final lighting sum. Instead of weighing lighting contribution by patch surface area, we compute a rudimentary average over the number of samples $|\Omega|$.

Mathematically, this approach is described by the equation for raytracing, as given in (2.25):

$$L_o(x,\vec{\omega}) = L_e(x,\vec{\omega}) + \frac{1}{|\Omega|} \sum_{\vec{\omega}_i \in \Omega} f_r(\vec{\omega}_i,\vec{\omega},x) L_o(I(x,\vec{\omega}_i),-\vec{\omega}) \frac{\vec{\omega}_i \cdot \vec{n}_x}{\|x - I(x,\vec{\omega}_i)\|^2} \quad (5.7)$$

Since radiosity is only concerned with diffuse reflections, the BRDF f_r collapses into a generic diffuse BRDF of $\frac{\rho}{\pi}$:

$$L_o(x) = L_e(x) + \frac{\rho}{\pi|\Omega|} \sum_{\vec{\omega}_i \in \Omega} L_o(I(x,\vec{\omega}_i)) \frac{\vec{\omega}_i \cdot \vec{n}_x}{\|x - I(x,\vec{\omega}_i)\|^2} \quad (5.8)$$

The equation above can be adapted fairly easily into our existing architecture by using DXR's *closest-hit* shader which, in essence, corresponds to our intersection function $I(x,\vec{\omega})$.

The closest-hit shader allows one to retrieve information on the closest intersection, including UV coordinates which lets us access all required data through our existing texturegroup. The lighting value sampled by each ray can be blurred by sampling a higher mipmap level in order to avoid sporadic illumination effects due to small texture details.

Algorithm 8 outlines the respectively modified RTLPass for this approach. For the sake of brevity, we kept the exact operations (such as mipmap-sampling, accounting for patch-size, tangent-space transformation etc.) obfuscated, but the entire closest-hit shader can be viewed on the RTRad open source repository (see [63]).

Algorithm 8 RTLPass

1: **for** $i \in [0, n]$ **do**   ▷ For each patch (executed in parallel)
2:   $L_{out}(i) \leftarrow L_e(i)$ ▷ Set initial lighting value
3:   **for** $\vec{\omega} \in \Omega$ **do**
4:    $\vec{\omega}_t \leftarrow \vec{\omega}$ in tangent-space of patch i
5:    Shoot a ray from $pos(i)$ in direction $\vec{\omega}_t$
6:    **if** the ray hits another patch j **then**   ▷ Closest-hit shader
7:     $g \leftarrow \frac{\vec{\omega}_i \cdot nrm(i)}{\|pos(i) - pos(j)\|^2}$ ▷ Geometric factor
8:     $L_{out}(i) \leftarrow L_{out}(i) + g * mat(i) * \frac{\rho}{\pi|\Omega|} * L_o(j)$ ▷ Add contribution
9:    **end if**
10:   **end for**
11: **end for**

5.4.1 Hemispheric Direction Generation

The algorithm above assumes that a set of directions Ω is given. To sample directions uniformly across a hemisphere, we utilize the same method that Laine et al. use in *incremental instant radiosity* [31] to distribute additional VPLs from a light-source.

Laine et al. represent samples as 2D points inside a unit circle (rather than points on a hemisphere) [78, 31], which can subsequently be projected onto a hemisphere using the following operation:

$$\begin{pmatrix} x \\ y \end{pmatrix} \longrightarrow \begin{pmatrix} x \\ y \\ \sqrt{1 - (x^2 + y^2)} \end{pmatrix} \tag{5.9}$$

As can be observed in fig. 5.11, evenly scattered points on a circle result in a lower density of vectors that strongly deviate from the surface's normal vector. This, in essence, induces the geometric cosine-term applied in sampling [2].

In incremental instant radiosity, the direction a new VPL is shot towards gets determined by finding the largest empty circle within the unit circle and placing a new sample at its center [31]. This helps maintain an even distribution of VPLs, without prior knowledge on how many samples will be generated.

Geometrically, the largest empty circle inside a sampled area must be centered at either

- a vertex in the Voronoi diagram that touches connects three Voronoi regions,

- the intersection between an infinite Voronoi endge and the bounding polygon or

- a vertex of the bounding polygon [31].

With last option being irrelevant when the boundary is a perfect unit circle [31].

Our algorithm, which pre-computes a set of evenly distributed samples Ω, is largely based on this principle by adhering to the steps listed in algorithm 9.

---

[2]Our implementation follows the pseudo-code listed in algorithm 8, meaning we apply the geometric cosine-term (dot product), *despite* the projection onto a hemisphere already inducing the same effect. Leaving this factor out may produce different, potentially better, results.

Algorithm 9 Directional Sample Generation

1: $\Omega \leftarrow$ 4 random points in a unit circle
2: **for** n iterations **do**
3:     Calculate the Voronoi diagram of Ω
4:     $V \leftarrow$ Set of all Voronoi vertices inside the unit sphere as well as all intersections between Voronoi edges and the unit circle boundary.
5:     Find the point in V that has the largest distance to its nearest neighbour and add it as a new sample to Ω.
6: **end for**

The results of this algorithm can be observed in fig. 5.10, whilst fig. 5.11 shows their respective 3D directions. Incremental instant radiosity is intended to maintain relatively even distributions under a growing sample count. We chose this method to ensure users can select the exact amount of samples to be used for radiosity, whilst not having to worry about those samples being unevenly distributed.

In order to cut down on unnecessary computations during runtime, we use a separate program written in *Python* (included in the RTRad repository [63]) to pre-compute a list of directions, which are then simply read by the RTLPass shader in the exact same order they were added to the set.

Coding directions in this static manner allows us to omit the complexities of generating Voronoi diagrams during runtime, thus greatly increasing performance. The resulting downside is that the maximum amount of samples becomes restricted to the amount that was pre-generated, as well as the maximum DirectX array size. Our chosen upper limit was 1024, although much larger sets could theoretically be accommodated using GPU data-buffers.
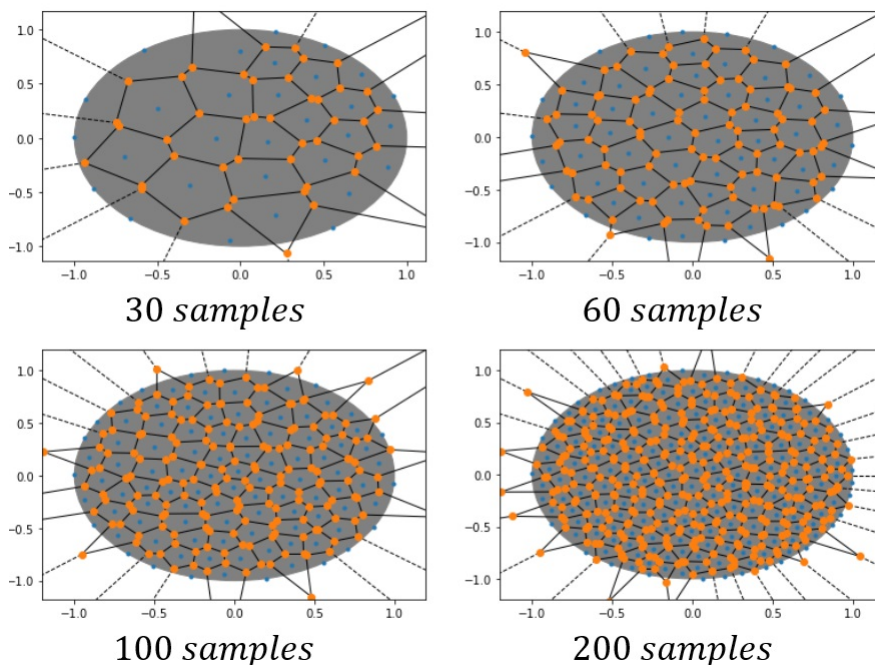

FIGURE 5.10: Unit circles with various amount of samples (blue) alongside their respective Voronoi diagrams (black/orange).
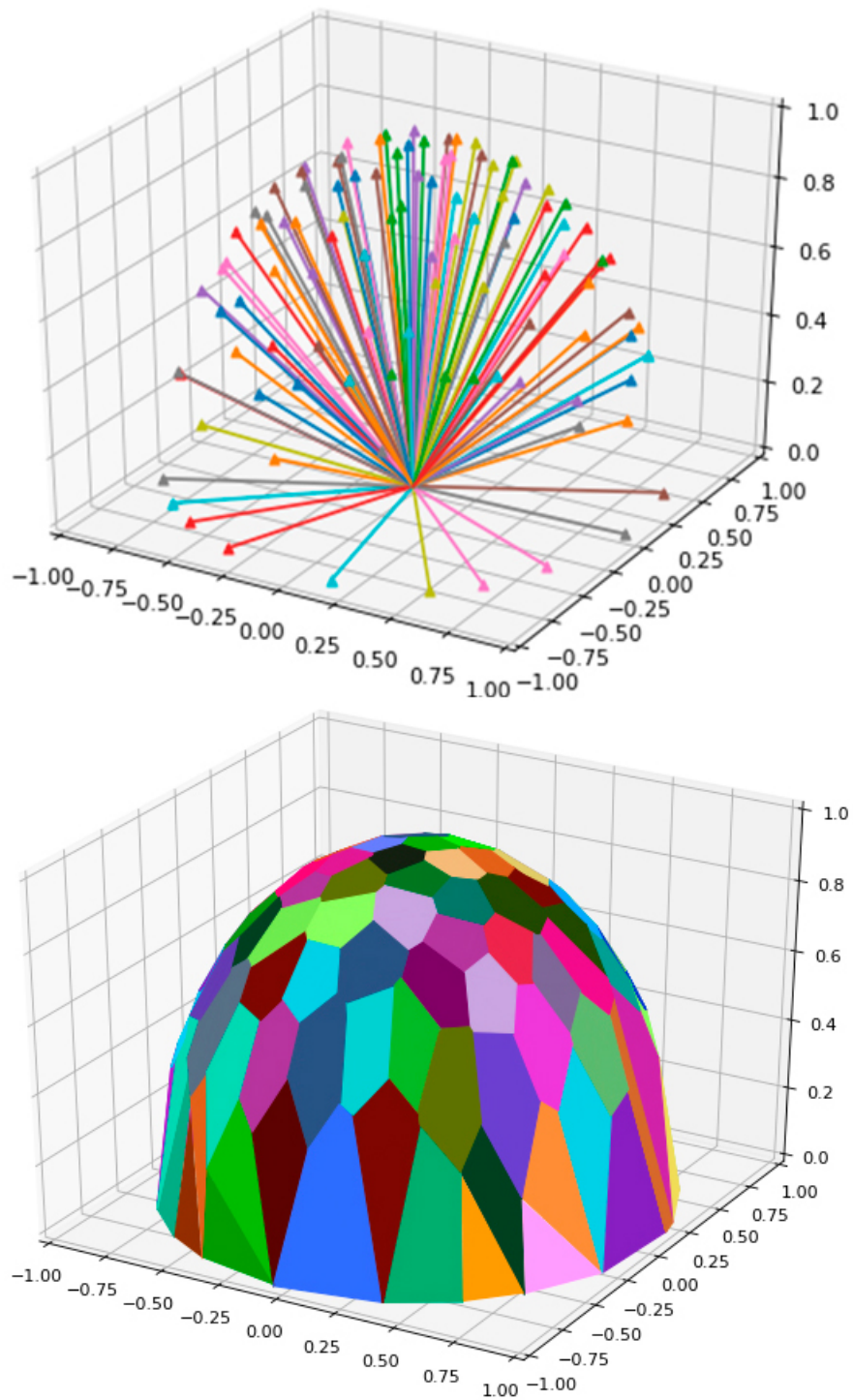
FIGURE 5.11: One hundred generated directions (top) and their corresponding Voronoi diagram projected onto a hemisphere (bottom).

Chapter 6

Evaluation

The advent of the Nvidia RTX platform allows us to offload the costly visibility computations of radiosity onto RT cores. By combining RTX and progressive refinement radiosity we seek to provide a competitively fast algorithm that does not compromise on visual fidelity.

This chapter aims to determine if and to what degree RTX can accelerate existing GPU radiosity implementations, as well as which quirks and enhancements are most beneficial. We will discuss our findings and assessments of the presented implementation and compare its performance with other industry-standard lightmap generators.

6.1 RTRad Overview

In chapter 4 we presented our implementation of an RTX-based progressive radiosity lightmap generator, which was subsequently expanded upon in chapter 5.

The proposed algorithm operates entirely on textures with a series of swift pre- and post-processing rasterization passes that translate the scene into a usable format, generate meta-information on refinement quad-trees and ameliorate leaks on UV seams, as shown in fig. 6.1.
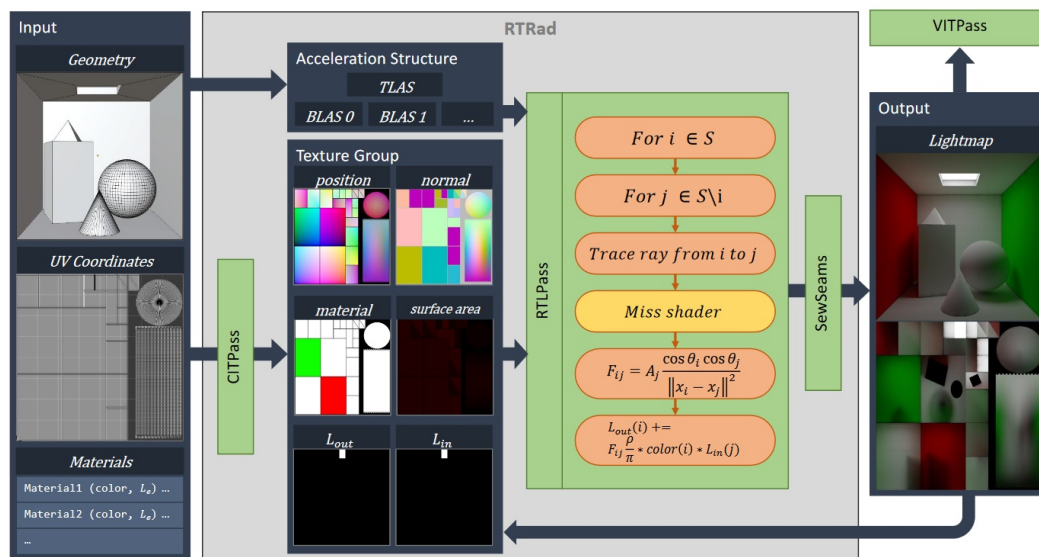

FIGURE 6.1: Simplified overview of the RTRad rendering passes. For a more detailed overview refer to fig. 4.9.

The most important component of our pipeline is the *RTLPass*, which is executed for each patch and uses the `TraceRay` function to test for visibility of other patches. A single-sample Monte-Carlo approximation of the view factors provides the lighting contribution for any patch that passes the test. Once completed, the input and output lighting textures are swapped for the next pass.

The RTLPass can be configured in manifold ways, including visibility caching, directional sampling and various methods for reducing sample counts. Fig. 6.2 shows an overview of these various configurations.
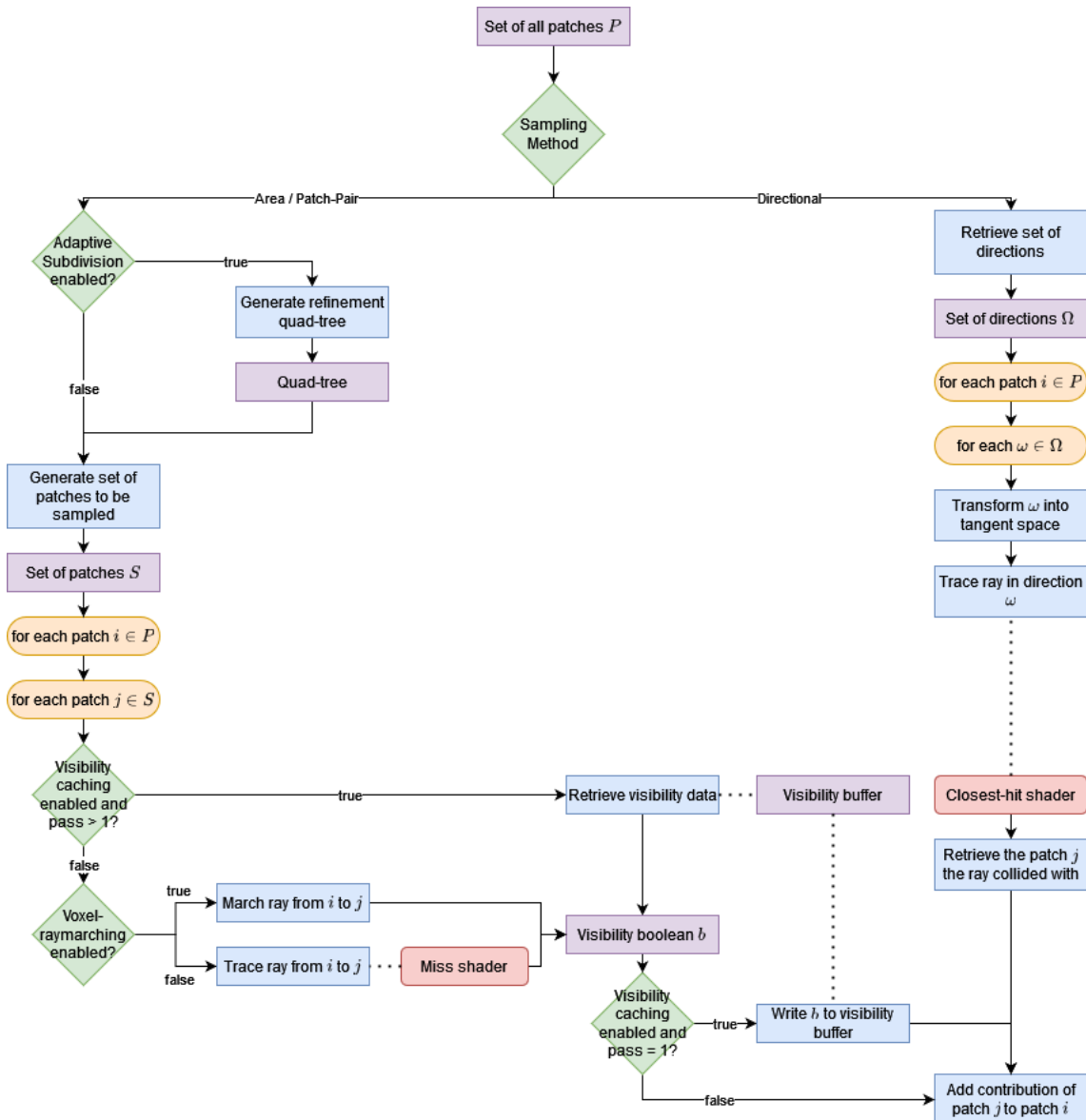
FIGURE 6.2: Flow-diagram of the RTLPass. Most condition branches are implemented using preprocessor directives and do not cost computation time.

6.2 Evaluation Method

The measurements we exhibit in this chapter were exclusively performed with the built-in Falcor profiler, which uses the most accurate available CPU and GPU timers to measure given functions as an event hierarchy [62].

Our benchmarking system consists of an AMD Ryzen 3900X CPU and an RTX 2070 Super graphics card[1] running on Windows 10.0.19044 with Nvidias GeForce Driver version 516.40.

6.2.1 Pass-Time

Our primary reference for performance is the *pass-time*, which constitutes the total of GPU time utilized by the RTLPass for all batches of a pass.

Our application automatically extracts this information from the profiler and outputs it at the end of a pass. We generally deemed pass-times of under a second to be suitable for real-time applications.

6.2.2 DFPR

To assess general lighting quality, this chapter provides images at the primary points of contention. In addition, we employ a self-conceived unit for *deviation from pure radiosity*, henceforth shortened as *DFPR*. This value measures how close the results of an optimized radiosity variant are to its un-optimized counterpart:

$$dfpr(L) = \frac{1}{n} \sum_{i=0}^{n} \|L(i) - P(i)\| \tag{6.1}$$

where L is the lightmap in question and P is a lightmap of the same size, generated with pure, progressive radiosity for the same scene.

The DFPR of an image is computed as its euclidean distance from a correspondingly large lightmap computed with pure radiosity, averaged across all pixels. A "perfect" DFPR of zero would equate to the image being an exact copy, whilst a theoretical worst DFPR would be $\sqrt{3}$ (the average RGB distance between a completely white and a completely black image).

We found that a DFPR value of 0.025 served as a very conservative threshold at which differences became noticeable to a human observer.

Fig. 6.3 illustrates this concept with a number of examples: Optimized radiosity algorithms run significantly faster, but also deviate from the results of pure radiosity. This deviation is given by the DFPR and serves as a measurement for visual fidelity under a given lightmap resolution.

Theoretically, dividing the DFPR by the corresponding pass-time could serve as a general measure for quality per cost, but we found that this did not accurately reflect empirical results.

---

[1]The 2070S contains a total of 40 RT Cores. Higher end RTX GPUs contain up to 82 and would perform respectively faster. We expect that lower end cards, which start at 30 RT cores, would still provide comparable results.
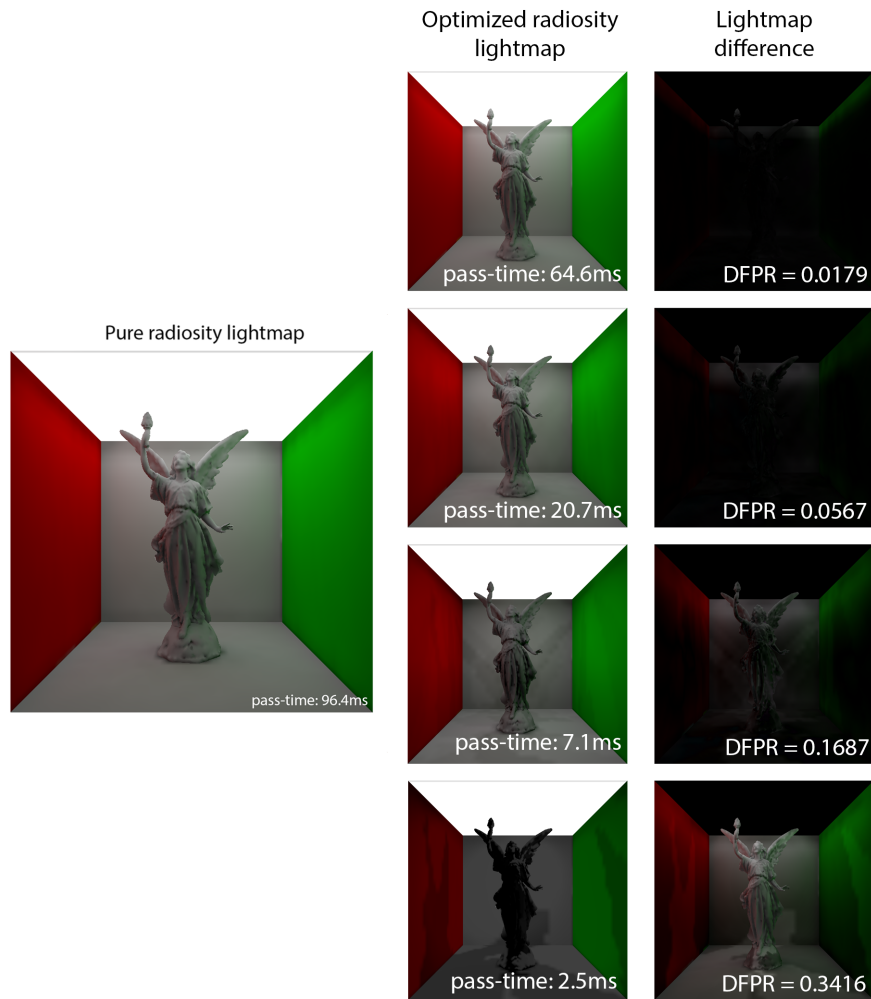
FIGURE 6.3: Examples of how the DFPR is calculated on lightmaps of varying quality.[2] The DFPR of a lightmap results from the average magnitude of all pixels in the subtraction of the lightmap itself and a corresponding "pure" lightmap.

6.2.3 Scenes

We employed a total of six different scenes for testing our algorithm and analyzing its performance, all of which are rendered through RTRad in fig. 6.4. Three of these are based on the *Cornell box* and are meant to directly demonstrate the system's capabilities in realistic global illumination. A further three scenes are significantly more complex and are meant to simulate a realistic use-case workload:

- **CornellLucy**: Consists of a simplified version[3] of the "Lucy" statue from the *Stanford 3D scanning repository* [64] inside a Cornell box with a large lightsource above. Used to test indirect light on a high-detail 3D model with abundant light.

---

[2] The lightmaps in this image are all of the size 128×128. The optimized variants are generated using Monte-Carlo undersampling with sampling windows of 2×2, 4×4, 8×8 and 16×16 respectively.

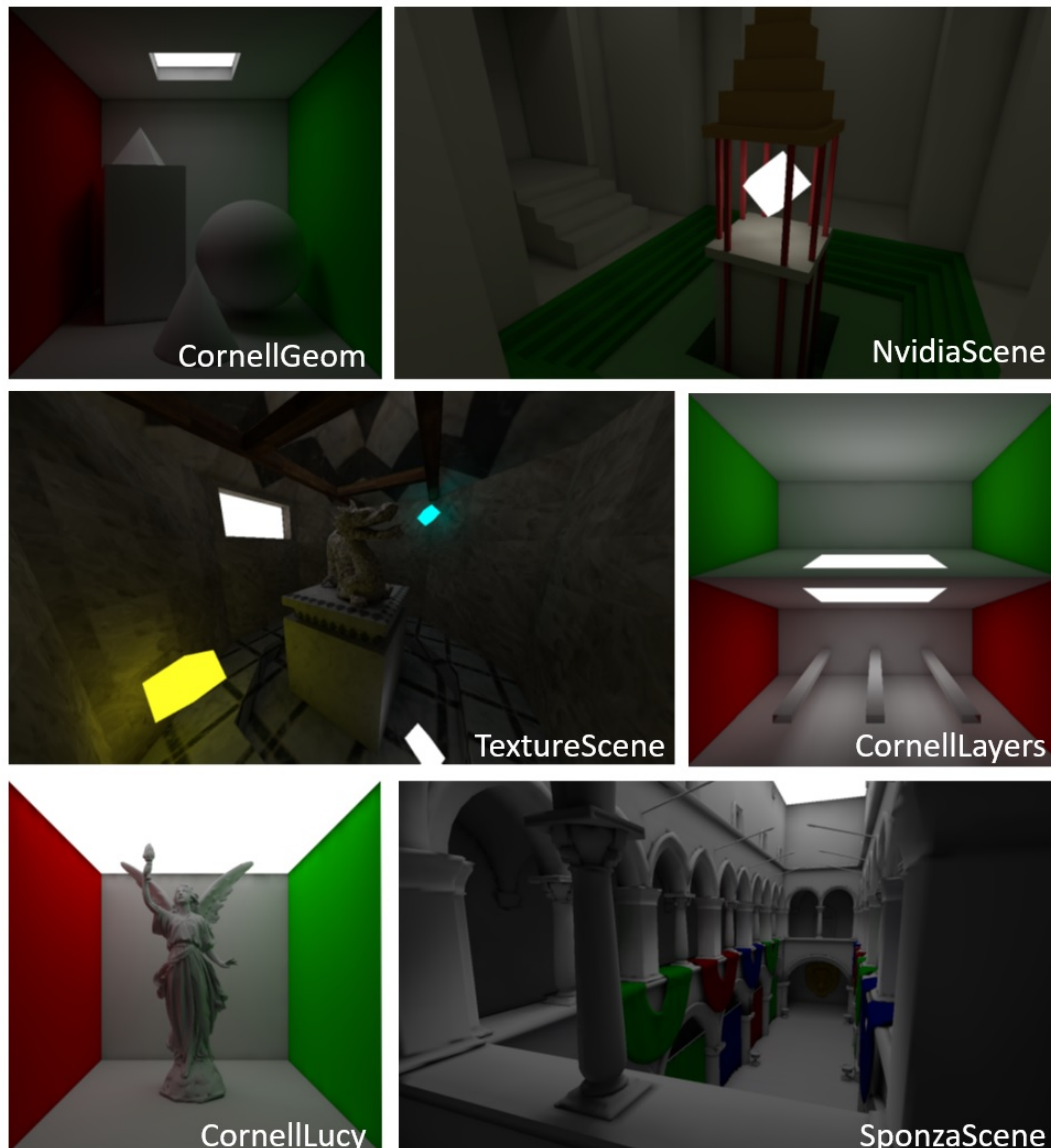[3] The same version that was used by Benjamin Kahl in VXCT [8].

FIGURE 6.4: Our employed testing scenes rendered through RTRad
with a lightmap of 1024^2 pixels.

- **CornellGeom**: Consists of a series of simple geometric shapes inside a Cornell box with a narrow light-source. Used to test inter-reflection in low-light environments and hard shadows.

- **CornellLayers**: A modified Cornell box that is split down the middle with a wedge. Used to ensure no color leaks from one section to the other as well as to test soft shadows.

- **SponzaScene**: A commonly employed graphics benchmarking scene created by *CryTek* [65]. To ensure compatibility with lower resolution lightmaps our version is somewhat simplified and texture-less, but still consists of over 150 thousand triangles.

- **NvidiaScene**: A reconstruction of the scene employed by Pharr et al. in the 2005 *GPU Gems 2* book [5]. It works well for testing small-scale light-sources as

well as a general point of comparison to the GPU-based progressive refinement solution presented by Nvidia in the aforementioned book.

- **TextureScene**: A custom-built scene that consists of several colored light-sources inside an octagonal room with a (simplified) dragon statue from the Stanford repository [64]. This scene is, uniquely, covered with detailed color-textures.

The overall triangle counts of these scenes range from from 60 (CornellLayers) to over 150 thousand (Sponza). Further scenes (with an adequate UV map) can be brought into RTRad quite easily in the form of FBX-formatted 3D models.

Based on Falcor's scene information, all acceleration structures consisted of a single TLAS, but were subdivided into multiple opaque BLAS geometries.

6.3 Results

6.3.1 Pure Progressive Radiosity

The core algorithm provided in chapter 4 constitutes an instance of progressive radiosity in of itself. This is a very crude, brute-force approach that does not compromise on realism but comes at a respectively high computational cost. A humbly sized lightmap of 256×256 pixels, for instance, would involve up to 256^4 visibility tests, resulting in well over 4 billion rays being traced.

This quadratic growth is clearly reflected in our measurements depicted in fig. 6.5. The approximate 4 billion rays are processed in approx. 2 seconds.

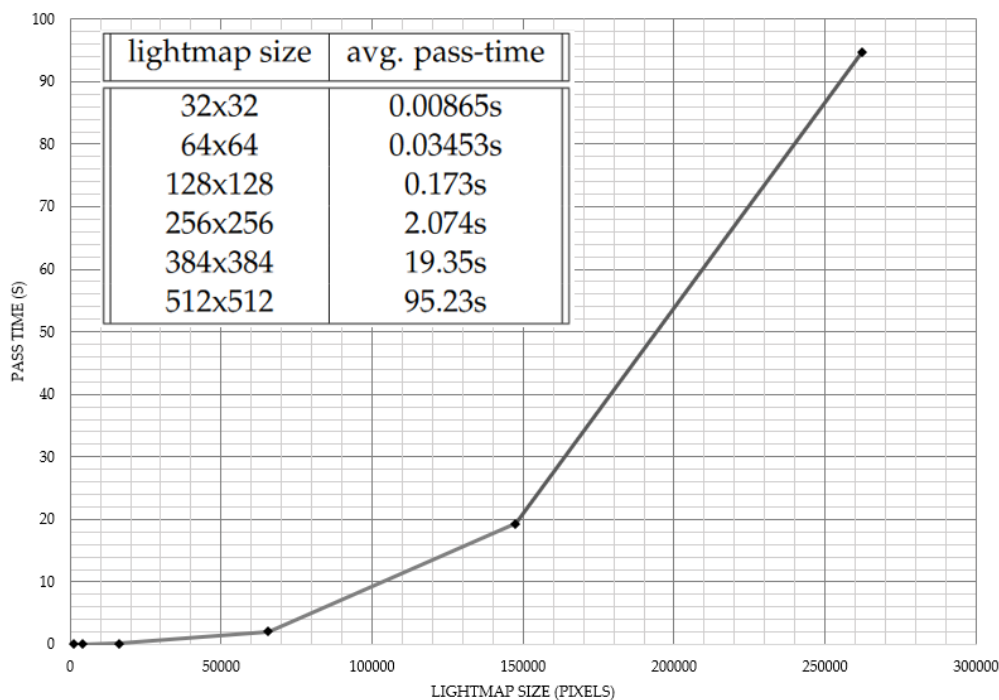| lightmap size	avg. pass-time
32x32	0.00865s
64x64	0.03453s
128x128	0.173s
256x256	2.074s
384x384	19.35s
512x512	95.23s

FIGURE 6.5: Average pass-time across three measurements of the same progressive radiosity workload for different lightmap sizes. The underlying scene (CornellLucy) has a UV mapping with a coverage of approx. 86%. The chosen batchsize is 64×64.

The pass-times required for low resolution lightmaps lie well within the leniencies of real-time applications, despite the costly nature of pure progressive radiosity.

Inevitably, the exponential cost overwhelms the leeway enabled by parallelization as resolutions reach the 512×512 mark[4], which is equivalent to an n of 260.000 (70+ billion visibility tests).

6.3.2 Undersampling

Although the scalability of quadradically complex algorithms can never be fully nullified, the cost of higher resolution lightmaps can be ameliorated by lowering the samples performed for each patch.
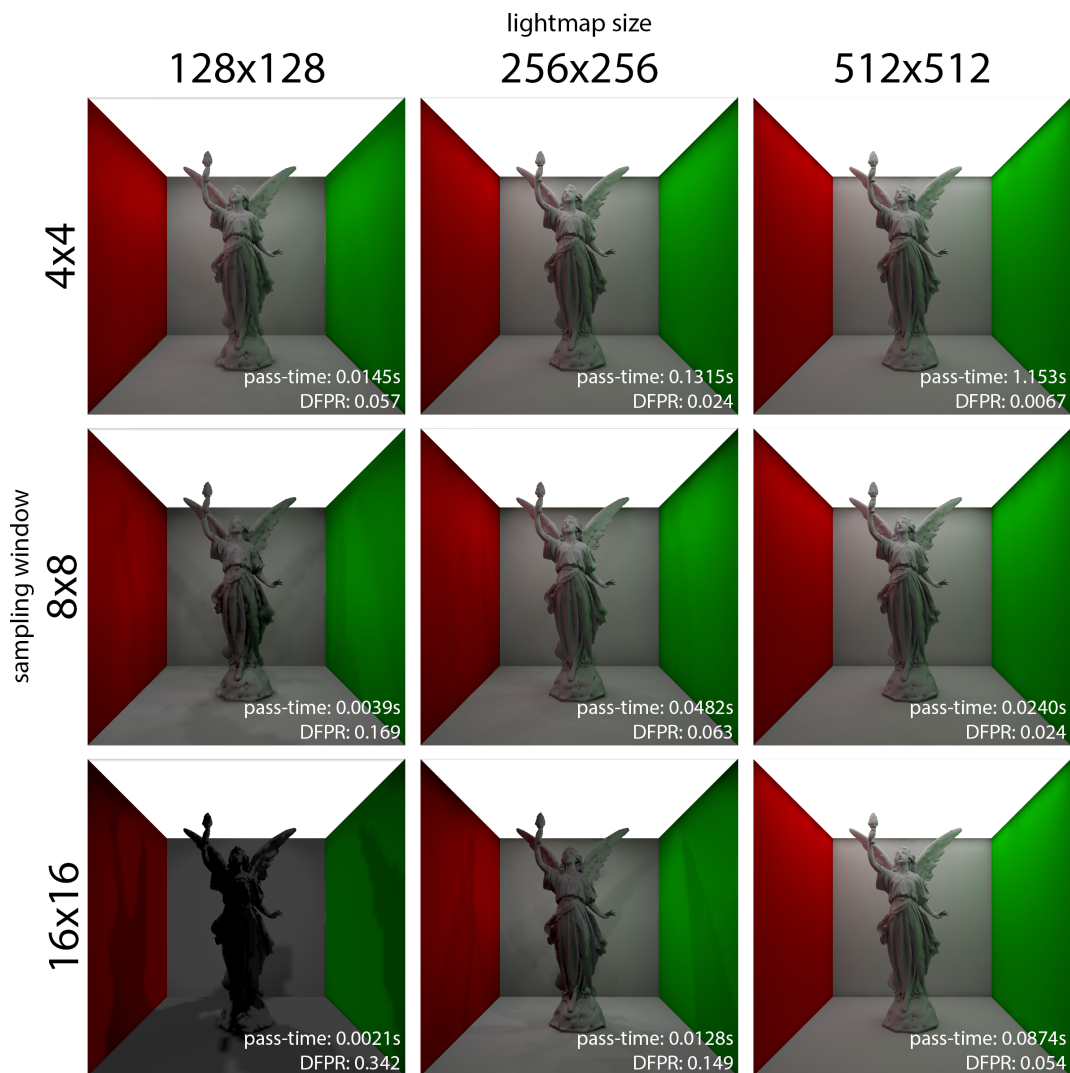
FIGURE 6.6: Results of Monte-Carlo undersampling after two passes for different combinations of texture resolution and sampling window. Anomalies only become noticeable when the sampling window is very large in relation to the lightmap size (towards the lower left corner).

---

[4]The largest measurement we took for pure progressive radiosity was a lightmap of 768×768 pixels, which took 867 seconds to complete.

Undersampling, as laid out in chapter 5, has an enormous positive impact on pass-times. Under the correct parameters, even larger lightmaps, such as 1024 × 1024, can be fully computed in under five seconds without any noticeable differences to their pure radiosity counterpart (DFPR < 0.025).

All three undersampling methods have near identical pass-times, except for Monte-Carlo undersampling, which requires 5 to 10 percent longer, presumably due to the large amount of pseudo-random number generation.

Differences in visual fidelity only become noticeable under extreme parameters where the sampling window is large in relation to the lightmap resolution (see fig. 6.6 and fig. 6.7). For smaller sampling windows the pass-time benefits massively whilst only producing minimal, indistinguishable differences in the final yield.

We found particular success with a ratio of $\frac{1}{128}$ between lightmap texture and sampling window, which produces adequate pass-times and DFPRs for most lightmap resolutions. In table 6.1 we list our measurements under these recommended settings.
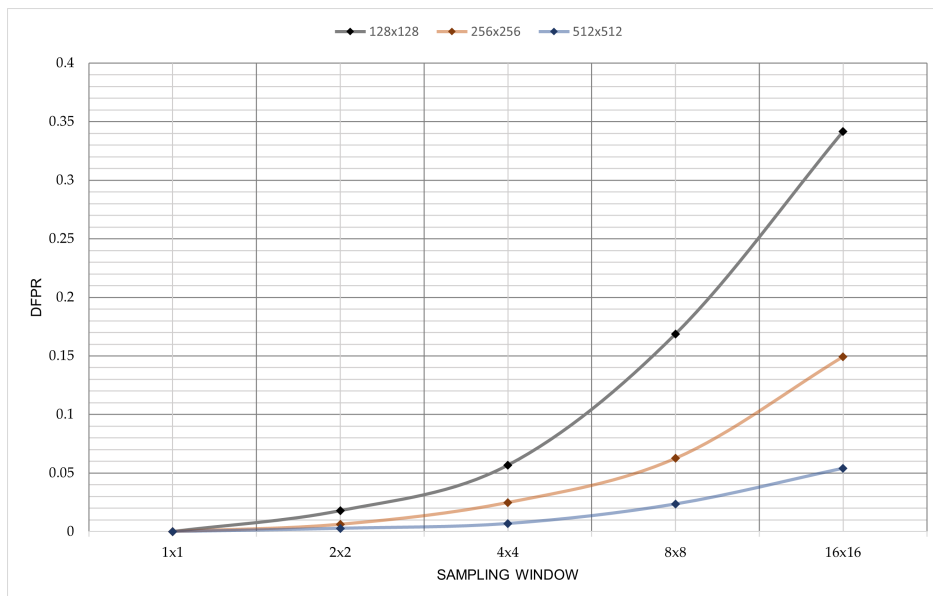

FIGURE 6.7: DFPR of Monte-Carlo undersampling for different resolutions and sampling windows (lower is better). Differences become noticeable at a threshold between 0.025 and 0.1, depending on the scene.

Lightmap Size	Sampling Window	DFPR	Pass-Time
128 × 128	1 × 1	0.0000	0.151s
256 × 256	2 × 2	0.0063	0.163s
512 × 512	4 × 4	0.0069	0.977s
1024 × 1024	8 × 8	<0.0196[5]	4.267s
2048 × 2048	16 × 16	<0.0345[5]	16.480s

TABLE 6.1: Pass-time and DFPR with Monte-Carlo undersampling using our recommended ratio of $\frac{1}{128}$.

Undersampling Method Differences

Despite being somewhat more costly, Monte-Carlo produces the highest quality lighting of the three undersampling methods. Whilst the differences are not substantial in most scenes, lightsources with small UV geometry can produce uneven shadows when lit with one of the alternatives.
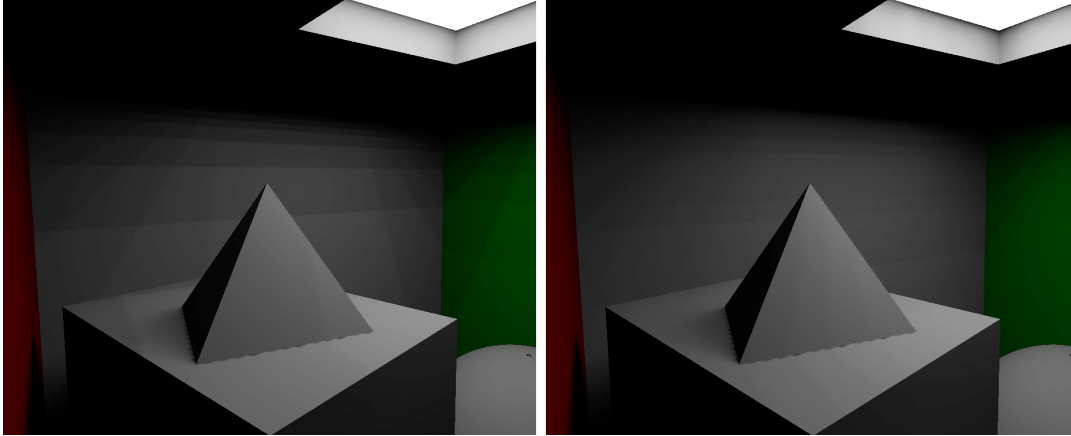

FIGURE 6.8: Uneven shadows produced by mipmapped or static-stride undersampling (left) as opposed to Monte-Carlo undersampling (right) in low-light environments.

The uneven shadows seen in fig. 6.8 are spread out by the randomization of Monte-Carlo undersampling which instead produces a minimal amount of noise (see fig. 6.9), unnoticeable under most circumstances.

We found that most other types of artifacts (such as unnatural highlights along corners) prominent in the latter two can almost entirely be eliminated by clamping the lighting contribution of any one patch to a maximum magnitude of 0.05.
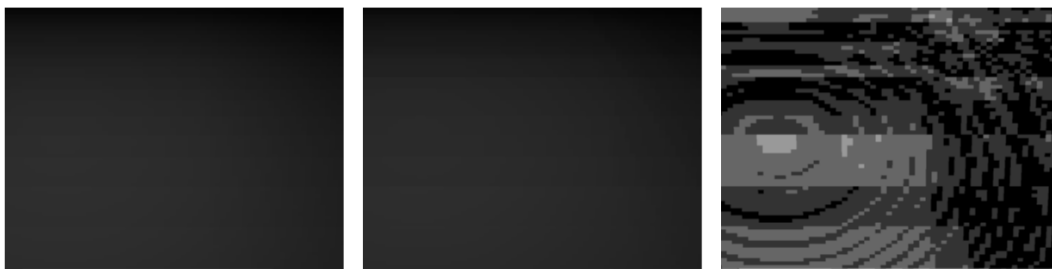

FIGURE 6.9: A single wall in CornellGeom lit with pure radiosity (left), Monte-Carlo undersampling (middle). To the right is the difference between the two textures, with its brightness multiplied by 50 to make the noise visible.

---

[5]We used a 512×512 lightmap as the pure reference to calculate this DFPR, which *overestimates* the value.

6.3.3 Adaptive Subdivision

Unlike undersampling, adaptive subdivision does not produce noise and is less prone to the cascading shadow effect highlighted in fig. 6.8.

The gains made in pass-time, however, largely depend on the gradient threshold that is chosen (see fig. 6.10).
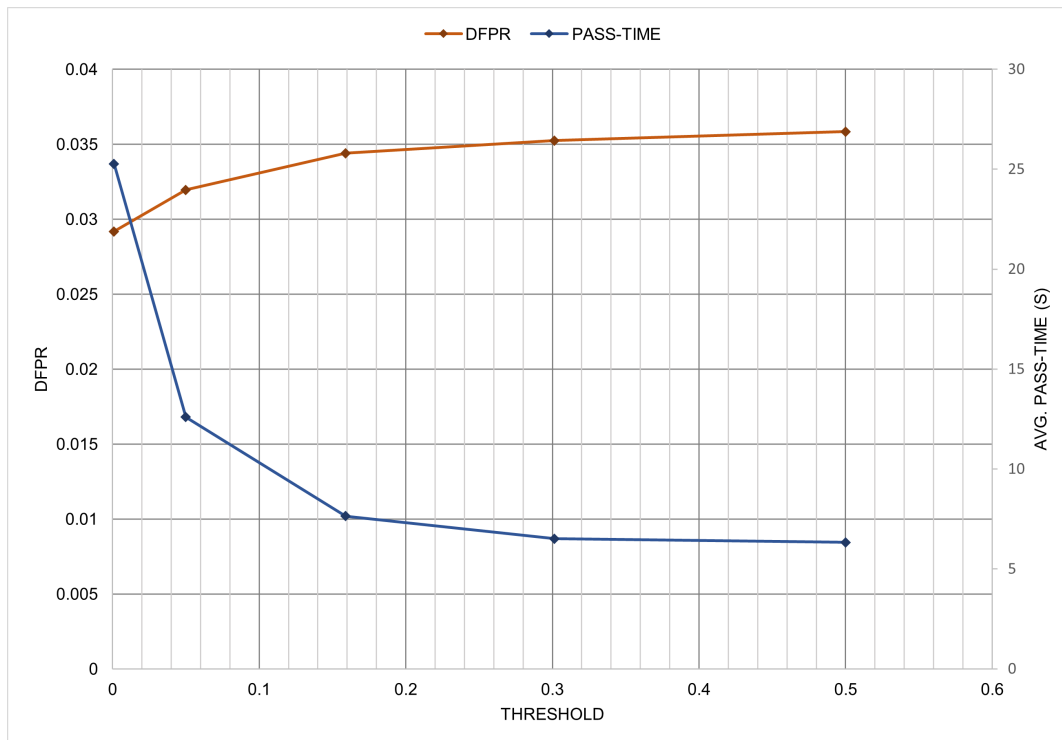

FIGURE 6.10: DFPR and average pass-time for different gradient thresholds in adaptive subdivision. Lightmap resolution is 512×512, max-node size 8×8 and batch-size 64×64.

Undersampling does not require the storage or retrieval of quad-tree metadata, and as such runs significantly faster in most cases. Rudimentary undersampling is simpler and its parameters are scene-independent, which makes the results more predictable. With adaptive subdivision even minor changes in the chosen gradient threshold can manifest themselves in significant differences in pass-times.

It is noteworthy that our approach is entirely contained within a textures alpha-channel, which allows for a simple implementation and visualization, but may offer worse performance than a quad-tree contained in a memory buffer or a separately generated sampling texture.

In general we found that adaptive subdivision, although offering vastly reduced pass-times over pure radiosity, gets frequently outclassed by the swifter pass-times offered by Monte-Carlo undersampling (see fig. 6.11).
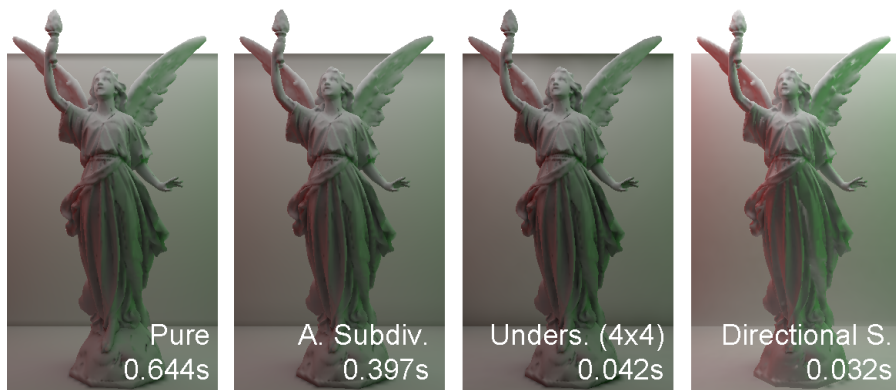
FIGURE 6.11: Avg. pass-time and visual comparison between different sampling techniques on a 256×256 lightmap. Lighting quality is worse with directional sampling, but pass-time scale linearly. Undersampling can exaggerate ambient occlusion, but is generally faster than adaptive subdivision.

6.3.4 Scene Complexity

Measuring performance relative to a scene's triangle count is difficult to do accurately, as the cost incurred by BVH traversal can highly vary depending on the specific arrangement and concentration of these triangles.

Nevertheless, our results are roughly indicative of the logarithmic scaling that the underlying theory would predict.

Fig. 6.12 depicts the pass-time for each of our scenes under the same pass parameters. The curve generally follows a logarithmic pattern, apart from our largest scene (SponzaScene) being significantly faster than our second largest (CornellLucy).
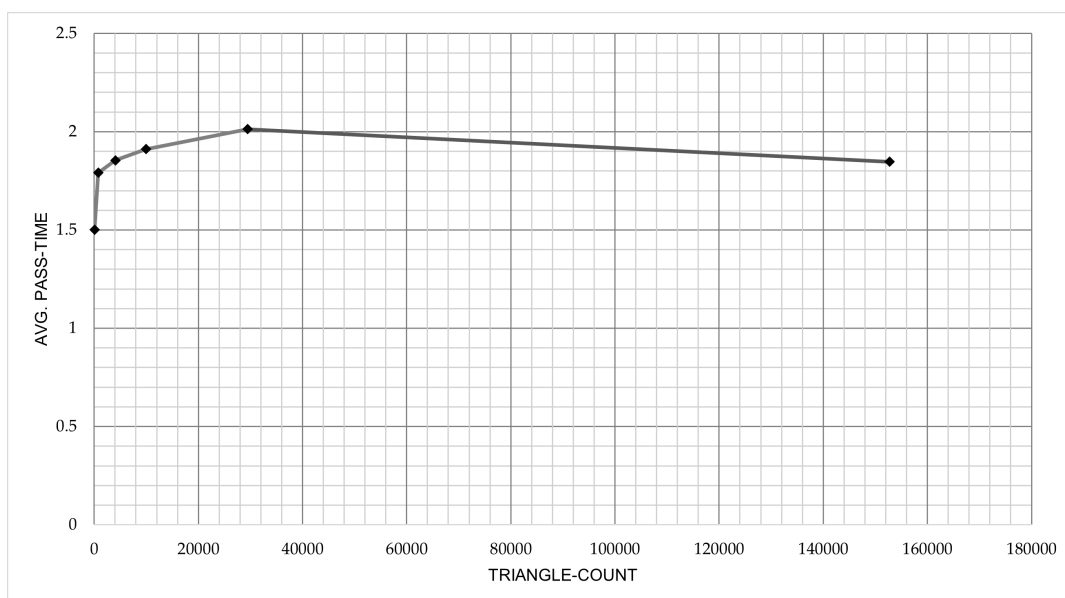

FIGURE 6.12: Average pass-time (across five passes) for each of our testing scenes. Parameters: 256×256, pure radiosity, batch-size of 64×64.

The underlying cause for this anomaly is likely the specific arrangement of the geometry in these scenes. CornellLucy has the vast majority of its triangles at its center, with the surrounding walls consisting of a large number of radiosity patches. Correspondingly, most rays cross this center area and will likewise be traced against large portions of the BVH.

SponzaScene on the other hand has its triangles spread out and segregated into smaller sub-volumes. Given that our rays are only as long as they need to be, it stands to reason that such an arrangement would induce a swifter BVH traversal per ray. Furthermore, RTX's ray-grouping technique [35] may contribute to this phenomenon as well.

6.4 Extensions

Below we disclose the impact provided by the implemented extensions that go beyond the scope of progressive refinement radiosity, such as visibility caching, ray-marching and directional sampling.

6.4.1 Visibility Caching

Visibility data can be stored in its entirety for lightmaps smaller or equal to 512×512 pixels. Correspondingly, all lightmaps of this size demonstrate a significant speedup for all passes beyond the first (see fig. 6.13). The first pass, in contrast, is slowed down by 10% to 20%, due to the necessity to compute the cantor-index and store the visibility bit.

Smaller lightmaps benefit less from this increase in speed, but the maximum size remains capped at 512×512.
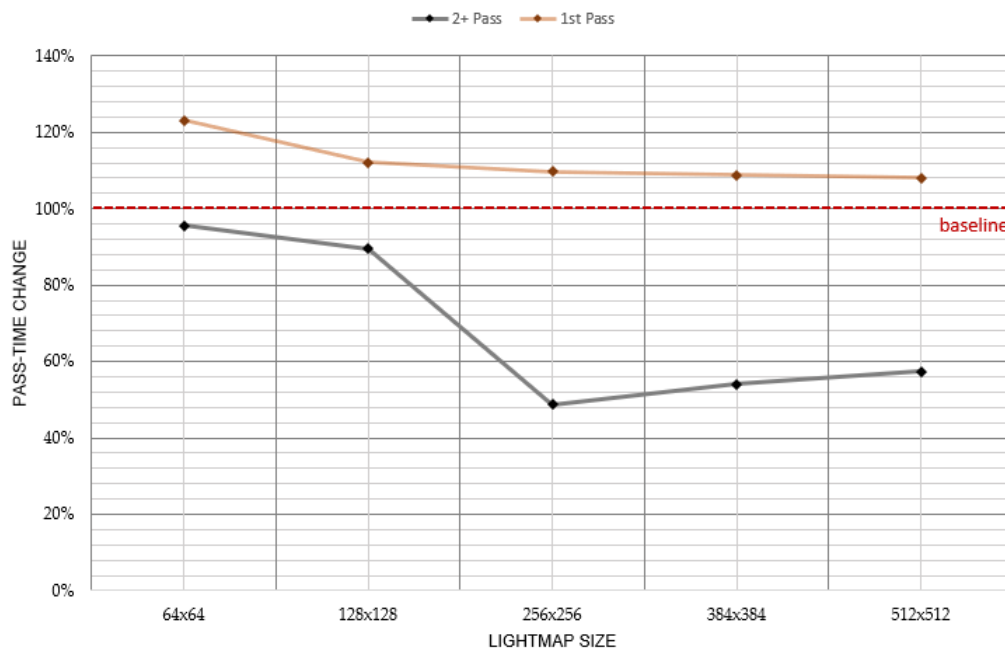

FIGURE 6.13: Percentage change in pass-time after enabling visibility caching for the first and subsequent passes. A value of 50% would equate to the pass running twice as fast, 200% twice as slow etc. (CornellLucy scene with maximum batch-size).

The overall computation speed is greatly improved for resolutions between 256×256 and 512×512. Unfortunately, higher resolutions cannot viably be covered by our method due to the respective buffer index exceeding the maximum value of an unsigned, 32bit memory pointer.

The unfortunate conclusion is thus that visibility caching is only *possible* for lower resolutions, which least require it. Although the approach may find itself useful for real-time lightmapping on small to medium sized lightmaps, the results demonstrate why GPU-based radiosity implementations generally refrain from caching visibility data: The available memory is unable to harbour the exponential amount demanded by larger lightmaps.

6.4.2 Voxel Raymarching

Visibility estimation through voxel-raymarching is inherently less accurate than RTX. Fig. 6.14 shows the precision of visibility estimation for a single patch, which is *favourable* to raymarching. Patches located in non-flat, detailed environments are likely to degrade accuracy even further, because all the detail within the vicinity gets simplified into a single voxel. This deterioration in quality can clearly be observed in fig. 6.15, where complex geometry, or geometry miss-aligned with the voxelmap produces unrealistic shadows.
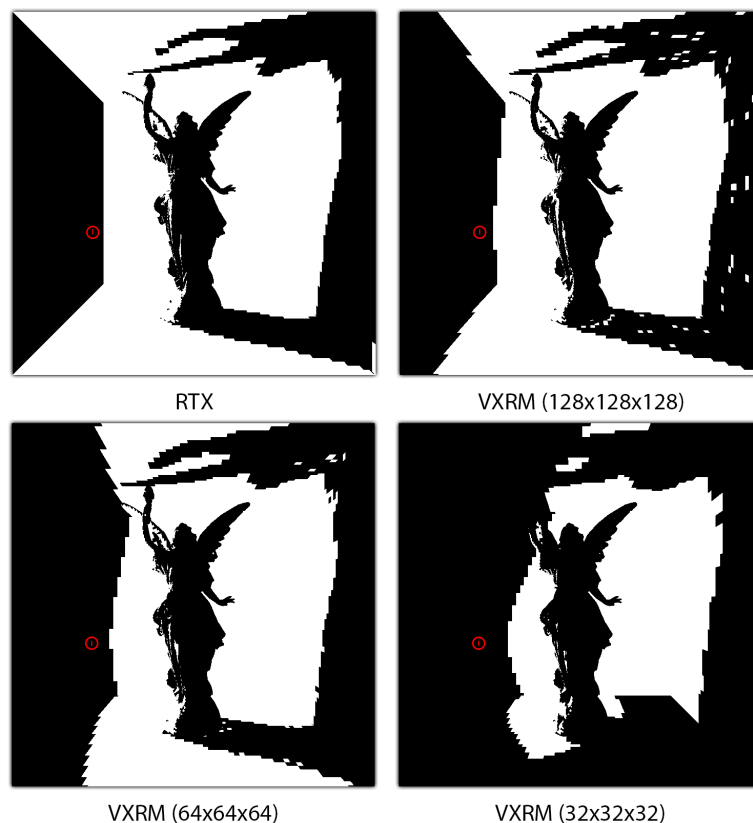

FIGURE 6.14: The red patch (highlighted by a circle) is visible to the white patches. Visibility accuracy of RTX compared with voxel raymarching on differently sized voxelmaps.
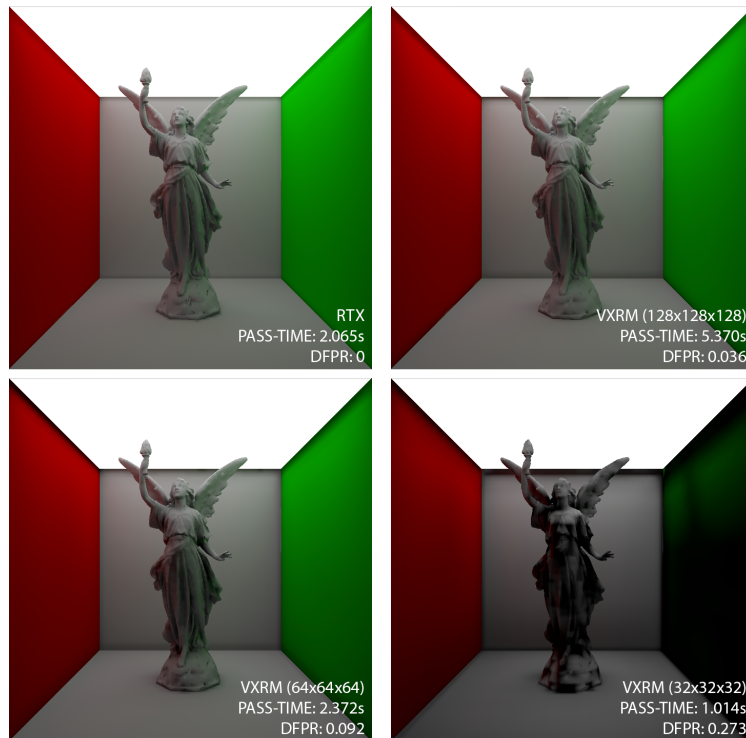
FIGURE 6.15: Pass-time and DFPR with regular RTX (left) and voxel raymarching on voxelmaps of different sizes. Pass-times only become worthwhile for very small voxelmaps, whereupon the quality degrades too much to remain viable.

In addition, voxel raymarches do not run on dedicated hardware and, as such, will not benefit from RT-core parallelization. In our testing, voxel raymarching was consistently slower than RTX-based raytracing, except for very small voxelmaps, which are less suited to complex scenes and provide unsatisfactory lighting quality.

In chapter 5 we postulated the idea of complementing our RTX-based algorithm with regular voxel raymarches to relieve pressure on the limited amount of RT cores.

Unfortunately, this concept did not materialize and ended largely in failure. The only instance for which interleaving ray-traces and raymarches demonstrated improved pass-times was when the voxel-maps were so small that raymarches were inherently faster.

However, using larger voxelmaps does provide accurate visibility and may serve as a fallback method for graphics cards that are not RTX compatible or in scenes that contain an exorbitant amount of triangles.

If nothing else, it is still serves as an indicative testament to RTX's impressive performance.

6.4.3 Directional Sampling

Employing a directional sampling approach as opposed to a rudimentary, patch-pair approach led to a dramatic improvement in performance for large lightmaps, even though the overall performance *per ray* suffered, due to rays traversing further into the BVH than previously.

This method is of linear complexity $O(n)$ relative to the amount of patches, as the number of rays fired per patch is constant, which makes it is highly adequate for

very large lightmaps. We were able to fully compute lightmaps of 2048^2 pixels in under two seconds, with 1024 samples taken per patch.

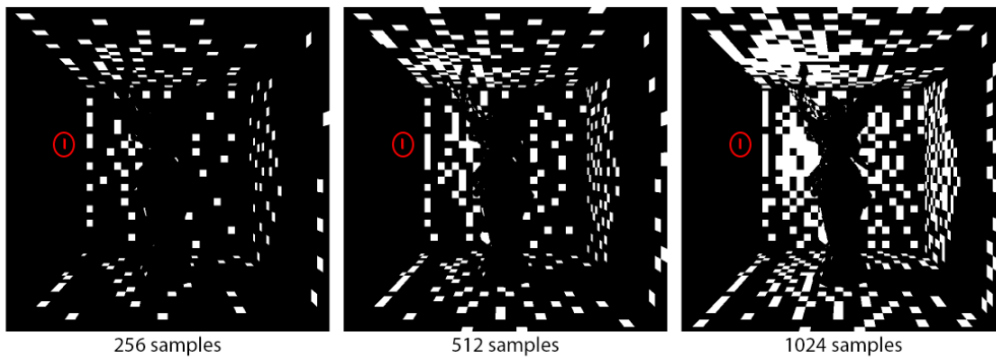256 samples · 512 samples · 1024 samples

FIGURE 6.16: Directional sampling with different amount of samples on a lightmap of 128×128 pixels. The white patches are sampled to compute the lighting value of the red patch (highlighted by a circle).

Visual fidelity is comparable with that of undersampling techniques, but significantly deteriorates in scenes with small light-sources, as seen in fig. 6.17.

Our implementation employs pre-computed directions that are hard-coded into the shader, which is very beneficial towards performance, but limits the maximum amount of samples to that of the pre-generated set. Examples of which patches our pre-generated sets sample can be seen in fig. 6.16.
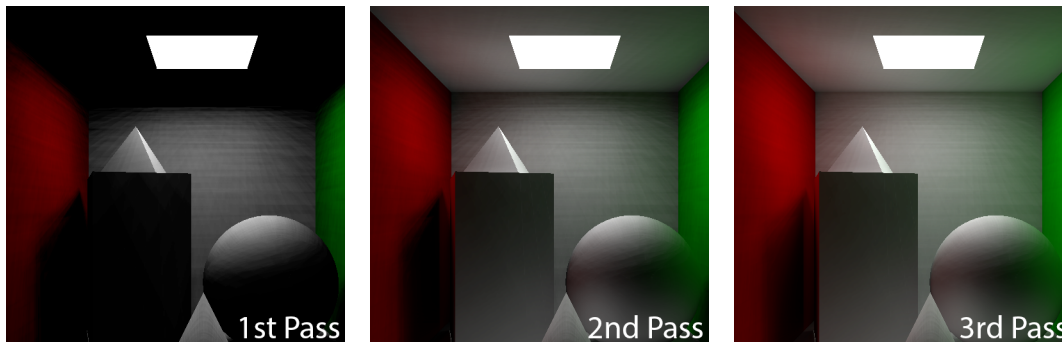

1st Pass · 2nd Pass · 3rd Pass

FIGURE 6.17: Directional sampling in environments with small light-sources. Because the chance of a ray hitting the light-source is low, noisy shadows are produced on the walls. The noise becomes less pronounced with each subsequent pass.

We recommend using directional sampling for large lightmaps in scenes with large light-sources, but suggest falling back to Monte-Carlo undersampling in other instances. Alternatively, a hybrid approach that utilizes directional sampling in conjunction with discrete sampling of important patches (such as light-sources) may provide an ideal middle-ground.
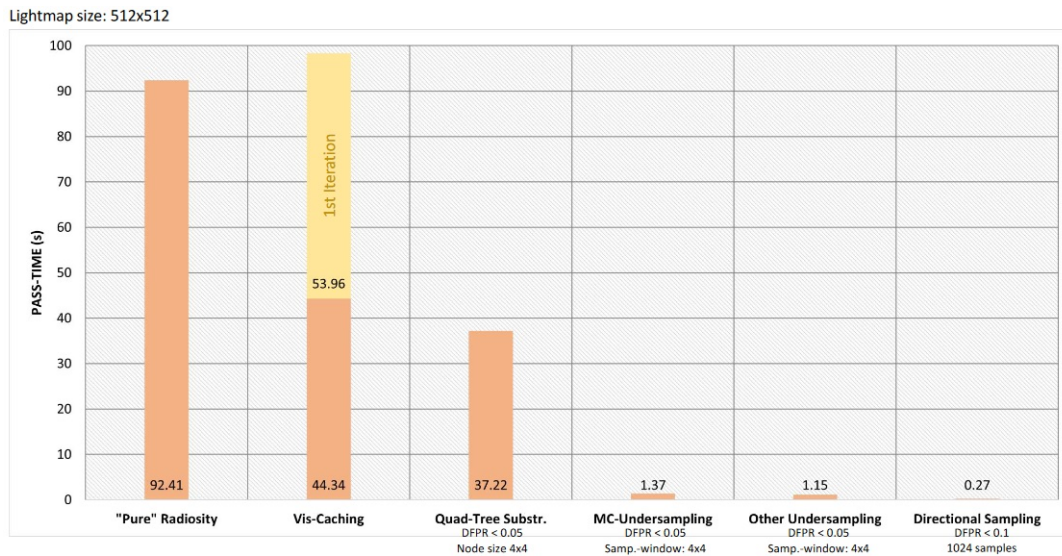
Lightmap size: 512x512

6.5 Comparison

In *GPU Gems 2*[6], Pharr et al. claim to achieve 2 frames per second on a scene with 10.000 patches using their GPU-based progressive refinement radiosity, with Coombe et al. achieving comparable results in their own implementation [17, 5].

In contrast, RTRad manages similar framerates with lightmaps of 60.000-100.000 patches. Fig. 6.19 provides a side-by-side comparison for visual differences between the two programs.
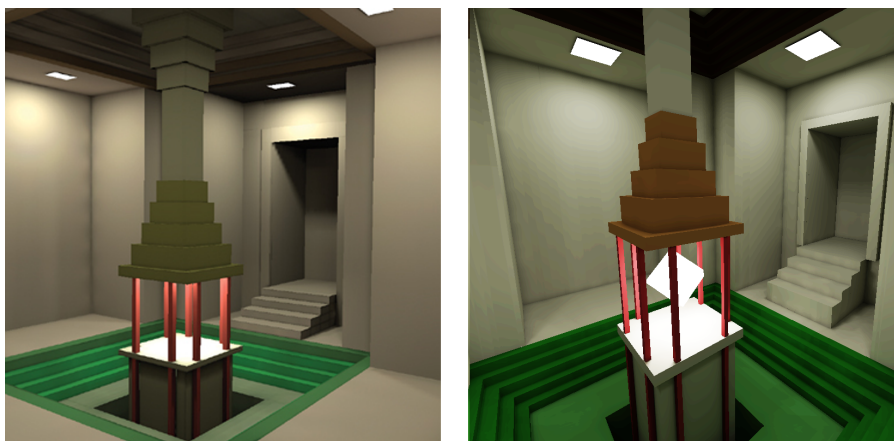

FIGURE 6.19: Scene with one million elements as shown in *GPU Gems 2* [5] (left) and our reconstruction rendered with RTRad (right).

---

[6]*GPU Gems 2* was published in 2005. Given the advancements in hardware since then, it is safe to assume that their system would run significantly faster on modern hardware.

6.5.1 Unity and Unreal Engine

Two of the most prolific rendering engines in the domains of video games, research and filmmaking, are *Unity* and *Unreal Engine* [45, 46].

Both of these have their own built-in CPU lightmap generation system: Unity in its *Progressive Lightmapper* and Unreal in its *Lightmass* system. Either system can also be set up for GPU execution[7].

Fig. 6.21 shows a performance comparison between each application , with a detailed account of this data listed in table 6.2. Despite being a purely research-oriented application that was developed with limited time and resources, RTRad competes impressively well with these well-established industrial solutions.
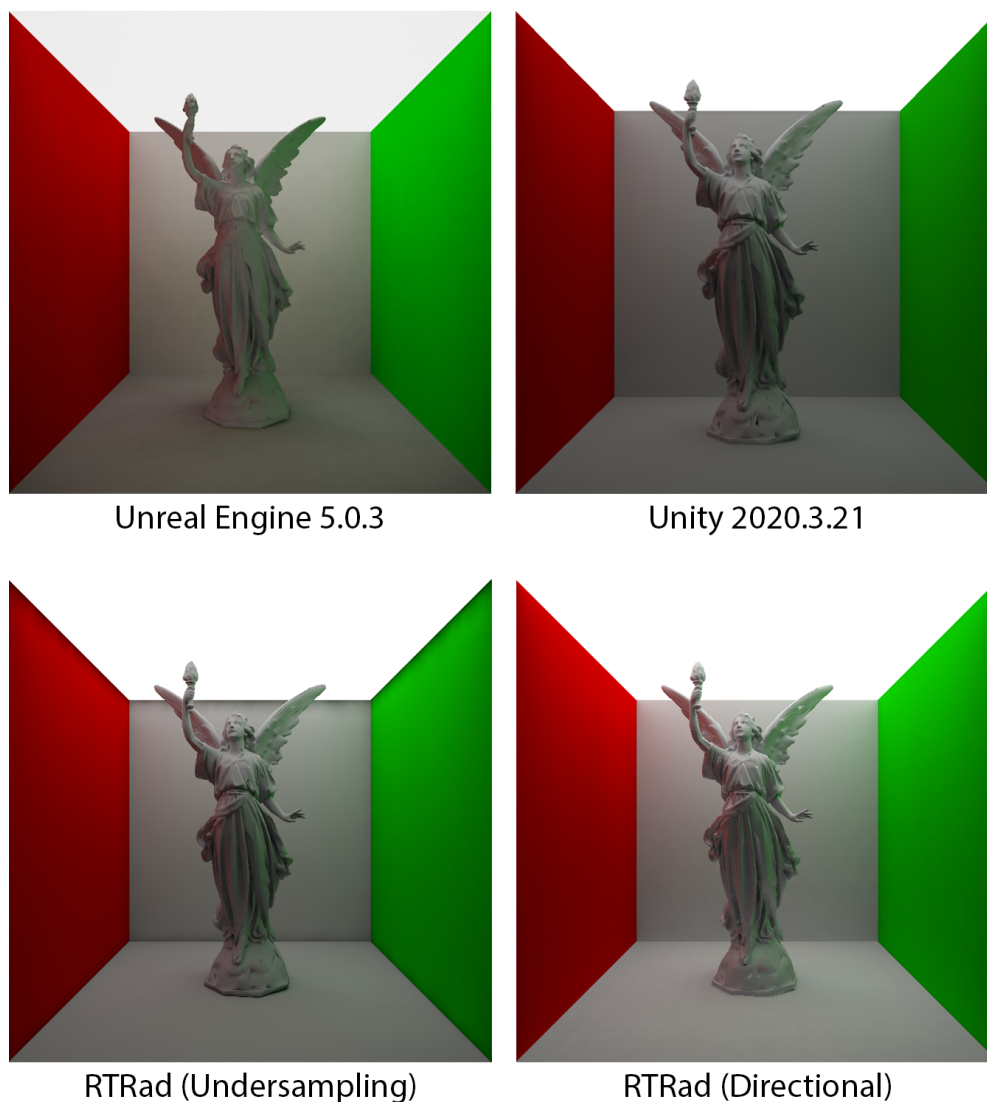

Unreal Engine 5.0.3

Unity 2020.3.21

RTRad (Undersampling)

RTRad (Directional)

FIGURE 6.20: Lightmap of 1024×1024 pixels for the same scene computed with different lightmapping tools.

---

[7]Unity's Progressive GPU Lightmapper and Unreal's GPU Lightmass are both preview/beta features that, whilst usable, have not yet been officially released.
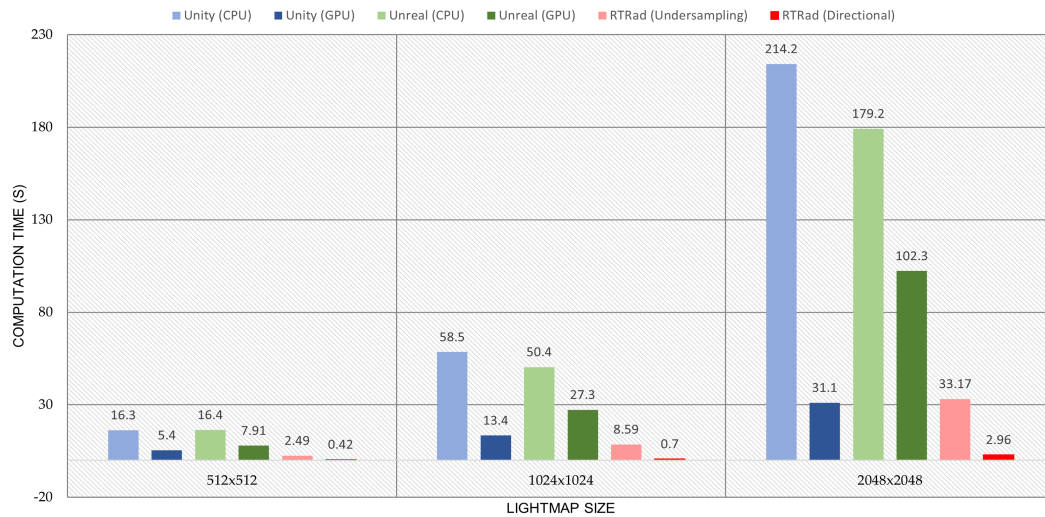
FIGURE 6.21: Total bake time of the CornellLucy scene for two bounces of light with different applications and lightmap sizes.

Lightmap Size	Device	Engine	Bake Time
256^2	CPU	Unity 2020.3.21	4.1s
256^2	GPU	Unity 2020.3.21	4.2s
256^2	CPU	Unreal Engine 5.0.3	6.5s
256^2	GPU	Unreal Engine 5.0.3	3.36s
256^2	GPU	RTRad (Undersampling)	0.5s
256^2	GPU	RTRad (Directional)	0.12s
512^2	CPU	Unity 2020.3.21	16.3s
512^2	GPU	Unity 2020.3.21	5.4s
512^2	CPU	Unreal Engine 5.0.3	16.4s
512^2	GPU	Unreal Engine 5.0.3	57.91s
512^2	GPU	RTRad (Undersampling)	2.49s
512^2	GPU	RTRad (Directional)	0.42s
1024^2	CPU	Unity 2020.3.21	58.5s
1024^2	GPU	Unity 2020.3.21	13.4s
1024^2	CPU	Unreal Engine 5.0.3	50.4s
1024^2	GPU	Unreal Engine 5.0.3	27.3s
1024^2	GPU	RTRad (Undersampling)	8.59s
1024^2	GPU	RTRad (Directional)	0.70s
2048^2	CPU	Unity 2020.3.21	214.2s
2048^2	GPU	Unity 2020.3.21	31.1s
2048^2	CPU	Unreal Engine 5.0.3	179.2s
2048^2	GPU	Unreal Engine 5.0.3	102.3s
2048^2	GPU	RTRad (Undersampling)	33.17s
2048^2	GPU	RTRad (Directional)	2.96s

TABLE 6.2: Expanded data corresponding to fig. 6.21.

To create comparable results (see fig. 6.20), each configuration was given its own custom settings, which we list below:

- *Unity Engine (CPU and GPU)*: Shadowmask, medium quality.

- *Unreal Engine (CPU)*: High quality preset.

- *Unreal Engine (GPU)*: 512 GI samples, no irradiance caching.

- *RTRad (Undersampling)*: Monte-Carlo undersampling in line with our recommended settings from table 6.1 and maximum batchsize.

- *RTRad (Directional)*: 1024 samples with maximum batchsize.

The exhibited measurements are naturally not all-conclusive, as Unity and Unreal are both powerful game engines made for far more than just lightmapping [56, 58]. Each algorithm employs diverging techniques and operates on entirely different parameters in addition to potential differences in their approach for time measurement.

We can, however, establish a clear trend in that leveraging RTX for radiosity is not only competitive, but highly advantageous in most circumstances.

6.6 Specular Reflections

A limitation inherent to radiosity as a whole is that only diffuse reflections are accounted for. RTRad does contain any non-diffuse functionality itself, but allows generated lightmaps to be exported as textures. These can then be brought into a different rendering pipeline to be complemented with specular reflections, as we did in fig. 7.1 with Unity.

Chapter 7

Verdict

In this chapter we sum up our primary conclusions in relation to our initial goals.

Additionally, we list some of the limitations encountered as well as questions that require further research to answer.

7.1 Summary

Radiosity has long proven itself an ideal solution for real-time global illumination in static scenes. Unfortunately, drawn-out periods of pre-computation can hinder productivity in designing 3D environments with realistic lighting.

Attempts to speed up this process have been centered around either

- the general reduction of computational complexity, as is done in instant radiosity, or

- the exploitation of its parallelizable nature.

Both of these techniques manifest themselves in *progressive refinement* radiosity, where patches are organized in quad-trees and updated simultaneously. Yet, few instances of this algorithm are implemented for GPU execution, and those that are typically rely on a z-buffered hemicube approximation for visibility, which comes at a significant expense whilst providing less realistic results.

Nvidia's Turing GPUs come equipped with dedicated raytracing hardware intended to speed up real-time raytracing. In chapter 2 we demonstrated how raytracing and radiosity both share the same underlying principles by deriving each method from the rendering equation, implying that the performance gains enabled by RTX ought not only to speed up raytracing, but also radiosity.

In chapter 6 we successfully demonstrated that this idea is both viable and competitive. Through our implementation we put a significant number of different methods and configurations to the test as well as examining additional extensions that serve as performance enhancements.

7.2 Limitations

In our evaluation we described the limitations encountered concerning visibility caching and voxel-raymarching in particular.

Since these do not present an obstacle to the core idea of leveraging RTX for progressive refinement radiosity, holistically, the concept appears viable.

Below we list our encountered limitations that are specific to our implementation of RTRad:

- Unless lightmap resolutions are sufficiently low or directional sampling is used, the performance remains marginally outside the domain of real-time.

- Directional samples either require large light-sources or a vast amount of rays to produce serviceable results.

- Our implementation of adaptive subdivision is subject to significant speed constraints because all pixels need to be tested for their alpha value. This problem may be ameliorated by creating an entirely separate texture consisting only of the pixels with an alpha value larger than zero.

- Our implementation only uses a single UV channel both for color textures and lightmaps. This approach is unsuitable for a PBR material system, but is easily expandable by including an additional channel.

- All light-sources must be reflected as a patch in the texturegroup. Point-lights or directional lights, as they are traditionally used in the phong illumination model, do not work.

7.3   Conclusions

Our analysis in chapter 6 comes with several implications. In regards to our initial goals, we arrived at the following overarching conclusions:

Conclusion 1: *On the GPU, simpler radiosity variants perform better.*
Performance-wise, our findings were generally more favourable to simpler techniques over adaptive subdivision. In our assessment, Monte-Carlo undersampling using the recommended settings listed in section 6.3.2 generally provides the best balance between speed, visual fidelity and reliability. For very large lightmaps and well-lit scenes, directional sampling is fastest.
Visibility-caching and voxel-raymarching are generally discouraged, but can be advantageous in specific use-cases.
We expect that a potential hybrid solution utilizing a conjoined set of patches sampled directionally and based on a quad-tree to provide ideal results.

Conclusion 2: *RTX can significantly improve GPU radiosity performance.*
As a whole, we were successful in demonstrating that RTX has the capability to speed up the offline lightmap generation process and, as more applications begin to move their heavy workloads onto the GPU, we expect radiosity to follow suit.
According to the *Steam Hardware Survey* of July 2022, approximately 30% of gaming-oriented PCs are equipped with a Turing graphics card [66]. If this number continues to grow, we anticipate engines utilizing GPU lightmapping (such as Unity, Blender or Unreal) to begin leveraging this power.

Conclusion 3: *RTX can quickly compute vast amounts of accurate visibility data for arbitrary point-pairs.*
Extensively, RTX appears to be a great potential asset to *any* application that requires large sets of highly accurate visibility data. This precedent is not strictly tied to just the domains of graphics or lighting, but may be expanded to computer vision, physics simulations and photogrammetry.

7.4 Future Work

RTX may prove useful for a variety of purposes depending on how widespread its adoption becomes. Our demonstration of its viability in lightmap generation opens up several new paths that would benefit from further research:

- We believe that the directional sampling approach in particular, which this thesis only touched on superficially, is the way forward for progressive radiosity on a very large scale. The potential that a hybrid approach may provide requires further examination. We expect that introducing multiple-importance sampling by prioritizing light-sources in combination with a number of directional samples for indirect light to function as a reasonable foundation for this approach.

- The utilization of *proxy geometry* has proved itself useful in computing global illumination with RTX [12]. It thus stands to reason that radiosity's performance may benefit from this as well.

- Whilst our implementation performs all calculations on the GPU, hybrid methods that combine both GPU and CPU have demonstrated potential in the past [37]. This opens the door to expanding a batch-wise view-factors computation on the GPU with RTX, whilst the CPU performs the steps related to light-transfer.

- Elias' radiosity implementation [77] utilizes a static texture that is multiplied with the samples of a hemicube. The brightness of this texture is proportional to the cosine of the angle between the surface normal, and the direction of the light which saves a significant amount of work on view factor calculation. Using a directional sampling approach with a hemicube and storing this value in the ray payload may prove to be faster and more flexible.

- Diffuse indirect light scatters uniformly and usually does not require the same resolution as direct light. A common practice in applications handling global illumination is to isolate direct and indirect light by computing them separately [5]. Using a model like Phong to compute direct light may allow a far lower lightmap resolution to be generated in RTRad to exclusively include the diffuse indirect component, enabling significantly shorter pass-times.
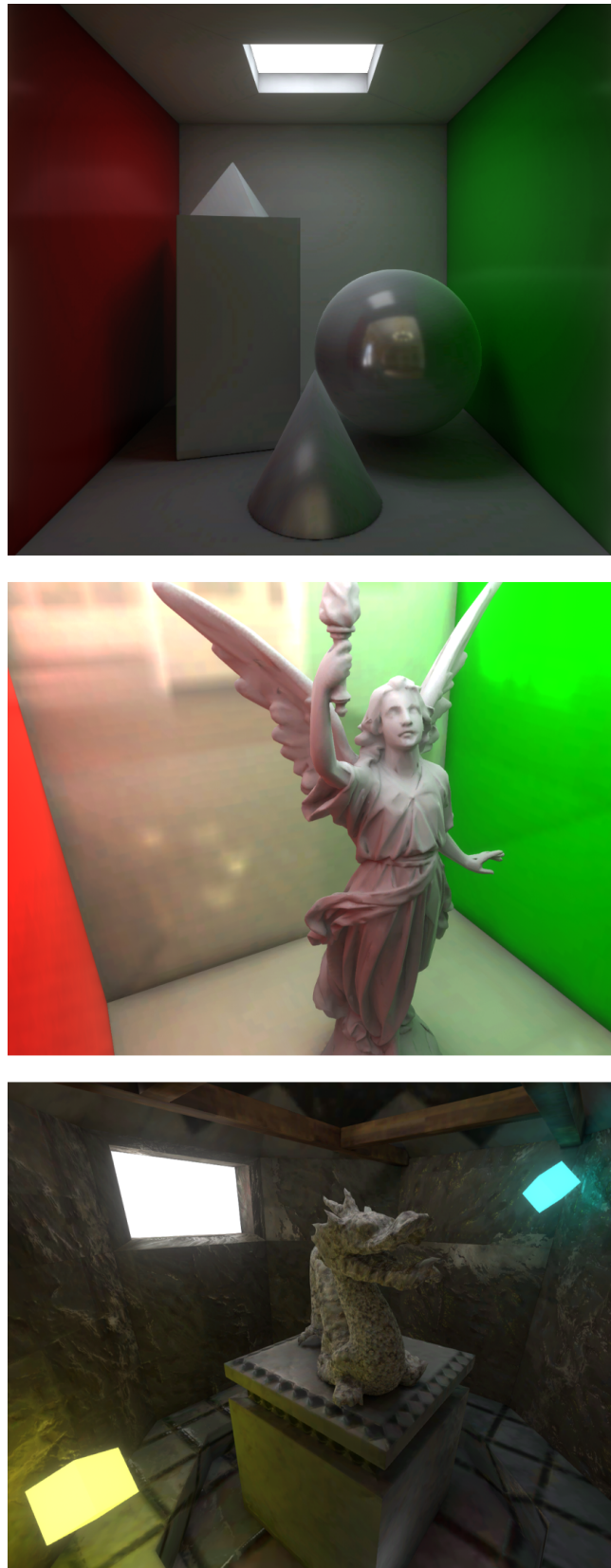
FIGURE 7.1: RTRad-generated lightmaps rendered in real-time using Unity's built-in rendering pipeline with specular reflection probes and some rudimentary post-processing effects.

Bibliography

Books

[1]   Michael F. Cohen, John Wallace, and Pat Hanrahan. *Radiosity and Realistic Image Synthesis*. San Diego, CA, USA: Academic Press Professional, Inc., 1993. ISBN: 0-12-178270-0.

[2]   R.G. Grainger. *Atmospheric Radiative Transfer (Draft Chapters)*. URL: http://eodg.atm.ox.ac.uk/user/grainger/research/book/ (visited on 09/30/2022).

[3]   Eric Haines and Tomas Akenine-Moller. *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*. USA: Apress, 2019. ISBN: 1484244265. DOI: 10.1007/978-1-4842-4427-2.

[4]   Alan Watt and Mark Watt. *Advanced Animation and Rendering Techniques*. New York, NY, USA: Association for Computing Machinery, 1991. ISBN: 0201544121.

[5]   Matt Pharr, ed. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison Wesley, 2005.

[6]   Ian Ashdown. *Radiosity: A Programmer's Perspective*. New York, NY, USA: John Wiley & Sons, Inc., 1994. ISBN: 0471304441.

[7]   Andrew S. Glassner. *Principles of Digital Image Synthesis*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994. ISBN: 1558602763.

Theses and Dissertations

[8]   Benjamin Kahl. *Real-Time Global Illumination Using OpenGL And Voxel Cone Tracing (Bachelor's Thesis) Freie Universität Berlin*. 2021. arXiv: 2104.00618.

[9]   Thomas Bernhard Koch. *Visibility Precomputation with RTX Ray Tracing*. [Diploma Thesis, Technische Universität Wien]. 2021. DOI: 10.34726/hss.2021.79729.

[10]   Joseph A. Shiraef. *An exploratory study of high performance graphics application programming interfaces*. Masters Theses and Doctoral Dissertations. May 2016. URL: https://scholar.utc.edu/theses/446/ (visited on 09/30/2022).

[11]   Maximilian Mader. *Hybrides Ray Tracing mit RTX-Technologie in Vulkan*. 2019. URL: https://nbn-resolving.org/urn:nbn:de:kola-18906 (visited on 09/30/2022).

[12]   Simon Moos. *Evaluating the Use of Proxy Geometry for RTX-based Ray Traced Diffuse Global Illumination*. 2020. URL: https://hdl.handle.net/20.500.12380/302019 (visited on 10/02/2022).

Articles and Proceedings

[13] James T. Kajiya. "The Rendering Equation". In: *SIGGRAPH Comput. Graph.* 20.4 (Aug. 1986), pp. 143–150. ISSN: 0097-8930. DOI: 10.1145/15886.15902.

[14] Arthur Appel. "Some Techniques for Shading Machine Renderings of Solids". In: *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference.* AFIPS '68 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, 1968, pp. 37–45. ISBN: 9781450378970. DOI: 10.1145/1468075.1468082.

[15] Derek Paddon and Alan Chalmers. "Parallel processing of the radiosity method". In: *Computer-Aided Design* 26.12 (1994), pp. 917–927. ISSN: 0010-4485. DOI: 10.1016/0010-4485(94)90057-4.

[16] Nathan Carr, Jesse Hall, and John Hart. "The Ray Engine". In: *Graphics Hardware 2002* (Jan. 2002), pp. 37–46.

[17] Greg Coombe, Mark J. Harris, and Anselmo Lastra. "Radiosity on Graphics Hardware". In: SIGGRAPH '05 (2005), 179–es. DOI: 10.1145/1198555.1198782.

[18] M. Young. "Pinhole Optics". In: *Appl. Opt.* 10.12 (Dec. 1971), pp. 2763–2767. DOI: 10.1364/AO.10.002763.

[19] Wojciech Jarosz et al. "Theory, Analysis and Applications of 2D Global Illumination". In: *ACM Transactions on Graphics (Presented at SIGGRAPH)* 31.5 (Sept. 2012), 125:1–125:21. DOI: 10/gbbrkb.

[20] Bui Tuong Phong. "Illumination for Computer Generated Pictures". In: *Commun. ACM* 18.6 (June 1975), pp. 311–317. ISSN: 0001-0782. DOI: 10.1145/360825.360839.

[21] Tomas Möller and Ben Trumbore. "Fast, Minimum Storage Ray/Triangle Intersection". In: *ACM SIGGRAPH 2005 Courses.* SIGGRAPH '05. Los Angeles, California: Association for Computing Machinery, 2005, 7–es. ISBN: 9781450378338. DOI: 10.1145/1198555.1198746.

[22] Cindy M. Goral et al. "Modeling the Interaction of Light between Diffuse Surfaces". In: *SIGGRAPH Comput. Graph.* 18.3 (Jan. 1984), pp. 213–222. ISSN: 0097-8930. DOI: 10.1145/964965.808601.

[23] John R. Wallace, Michael F. Cohen, and Donald P. Greenberg. "A Two-pass Solution to the Rendering Equation: A Synthesis of Ray Tracing and Radiosity Methods". In: *SIGGRAPH Comput. Graph.* 21.4 (Aug. 1987), pp. 311–320. ISSN: 0097-8930. DOI: 10.1145/37402.37438.

[24] Michael Cohen, Michael Greenberg, and P.C. Donald. "The Hemi-Cube: A Radiosity Solution for Complex Environments". In: *SIGGRAPH '85 conference proceedings. July, 1985. vol. 19 ; no. 3: pp. 31-39 : ill. (some col.). includes bibliography* 19 (Jan. 1985). DOI: 10.1145/325165.325171.

[25] Philippe Bekaert et al. "Hierarchical Monte Carlo Radiosity". In: *Proc. of the 9th Eurographics Workshop on Rendering* (Jan. 1998), pp. 259–268. DOI: 10.1007/978-3-7091-6453-2_24.

[26] Alexander Keller. "Instant Radiosity". In: SIGGRAPH '97 (1997), pp. 49–56. DOI: 10.1145/258734.258769.

[27] Cyril Crassin et al. "Interactive Indirect Illumination Using Voxel Cone Tracing". In: *Comput. Graph. Forum* 30 (Sept. 2011), pp. 1921–1930. DOI: 10.1111/j.1467-8659.2011.02063.x.

[28] Min-Zhi Shao and Norman Badler. "A Gathering and Shooting Progressive Refinement Radiosity Method". In: (Jan. 1993).

[29] Michael F. Cohen et al. "A Progressive Refinement Approach to Fast Radiosity Image Generation". In: *SIGGRAPH Comput. Graph.* 22.4 (June 1988), pp. 75–84. ISSN: 0097-8930. DOI: 10.1145/378456.378487.

[30] Michael F. Cohen et al. "An Efficient Radiosity Approach for Realistic Image Synthesis". In: *IEEE Computer Graphics and Applications* 6.3 (1986), pp. 26–35. DOI: 10.1109/MCG.1986.276629.

[31] Samuli Laine et al. "Incremental Instant Radiosity for Real-Time Indirect Illumination". In: EGSR'07 (2007). DOI: https://dl.acm.org/doi/10.5555/2383847.2383883, pp. 277–286.

[32] W Chin and S Ntafos. "Optimum Watchman Routes". In: SCG '86. Yorktown Heights, New York, USA: Association for Computing Machinery, 1986, pp. 24–33. ISBN: 0897911946. DOI: 10.1145/10515.10518.

[33] Jiří Bittner and Peter Wonka. "Visibility in Computer Graphics". In: *Environment and Planning B: Planning and Design* 30.5 (2003), pp. 729–755. DOI: 10.1068/b2957.

[34] Charley M. Wu et al. "Specialization and selective social attention establishes the balance between individual and social learning". In: *bioRxiv* (2021). DOI: 10.1101/2021.02.03.429553.

[35] Vadim Sanzharov et al. "Examination of the Nvidia RTX". In: (Nov. 2019), pp. 7–12. DOI: 10.30987/graphicon-2019-2-7-12.

[36] M Gatu Johnsson. "Approximating ray traced reflections using screenspace data". In: 2014.

[37] Eduardo D'Azevedo et al. "Solving a large scale radiosity problem on GPU-based parallel computers". In: *Journal of Computational and Applied Mathematics* 270 (2014). Fourth International Conference on Finite Element Methods in Engineering and Sciences (FEMTEC 2013), pp. 109–120. ISSN: 0377-0427. DOI: 10.1016/j.cam.2014.02.011.

[38] Alexandr Shcherbakov and Vladimir Frolov. "Dynamic Radiosity". In: (Jan. 2019). DOI: 10.24132/CSRN.2019.2901.1.10.

[39] Takeo Igarashi and Dennis Cosgrove. "Adaptive Unwrapping for Interactive Texture Painting". In: I3D '01 (2001), pp. 209–216. DOI: 10.1145/364338.364404.

[40] Andrew J. Willmott and Paul S. Heckbert. "An Empirical Comparison of Radiosity Algorithms". In: (Apr. 1997).

[41] Mike Houston and Wei Koh. "Compression in the Graphics Pipeline". In: (2001).

[42] Matthew P. Szudzik. "The Rosenberg-Strong Pairing Function". In: (2017). DOI: 10.48550/ARXIV.1706.04129.

[43] Kevin Watters and Fernando Ramallo. "Raymarching Toolkit for Unity: A Highly Interactive Unity Toolkit for Constructing Signed Distance Fields Visually". In: *ACM SIGGRAPH 2018 Studio*. SIGGRAPH '18. Vancouver, British Columbia, Canada: Association for Computing Machinery, 2018. ISBN: 9781450358194. DOI: 10.1145/3214822.3214828.

[44] Cyril Crassin and Simon Green. "Octree-Based Sparse Voxelization Using the GPU Hardware Rasterizer". In: OpenGL Insights. 2012.

[45] Eleftheria Christopoulou and Stelios Xinogalos. "Overview and Comparative Analysis of Game Engines for Desktop and Mobile Devices". In: *International Journal of Serious Games* 4.4 (Oct. 2017). DOI: 10.17083/ijsg.v4i4.194.

[46] A. Andrade. "Game engines: a survey". In: *EAI Endorsed Transactions on Game-Based Learning* 2 (Nov. 2015), p. 150615. DOI: 10.4108/eai.5-11-2015.150615.

[47] Alfred Schmitt, Heinrich Müller, and Wolfgang Leister. "Ray Tracing Algorithms — Theory and Practice". In: (1988). Ed. by Rae A. Earnshaw, pp. 997–1030. DOI: 10.1007/978-3-642-83539-1_42.

[48] Cyril Crassin et al. "Interactive Indirect Illumination Using Voxel Cone Tracing: A Preview". In: I3D '11. San Francisco, California: Association for Computing Machinery, 2011, p. 207. ISBN: 9781450305655. DOI: 10.1145/1944745.1944787.

Manuals and Documentation

[49] Nvidia Corp. *Nvidia Turing GPU architecture whitepaper (version WP-09183-001-v0)*. 2018. URL: https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf (visited on 09/30/2022).

[50] Microsoft Corp. *DirectX 12 Documentation*. URL: https://docs.microsoft.com/en-us/windows/win32/direct3d12/direct3d-12-graphics (visited on 09/30/2022).

[51] Nvidia Corp. *CUDA C++ Programming Guide (ver. PG-02829-001-v11.6)*. Mar. 2022. URL: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf (visited on 09/30/2022).

[52] Peter N. Glaskowsky. *NVIDIA's Fermi : The First Complete GPU Computing Architecture*. 2009. URL: https://www.nvidia.com/content/PDF/fermi_white_papers/P.Glaskowsky_NVIDIA%27s_Fermi-The_First_Complete_GPU_Architecture.pdf (visited on 09/30/2022).

[53] Mark Segal and Kurt Akeley. *OpenGL 4.6 Core Profile*. Oct. 2019. URL: https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf (visited on 09/30/2022).

[54] Nvidia Corp. *Nvidia Documentation on RTX Extensions for Vulkan and OpenGL*. URL: https://developer.nvidia.com/vulkan-turing (visited on 09/30/2022).

[55] Microsoft Corp. *DirectX Raytracing (DXR) Functional Spec (v1.19 8/10/2022)*. URL: https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html (visited on 09/30/2022).

[56] Epic Games Inc. *Unreal Engine 5 Documentation*. 2022. URL: https://docs.unrealengine.com/5.0/en-US/ (visited on 10/01/2022).

[57] Epic Games Inc. *Unreal Engine 4.27 Documentation - Unwrapping UVs for Lightmaps*. URL: https://docs.unrealengine.com/4.27/en-US/WorkingWithContent/Types/StaticMeshes/LightmapUnwrapping/ (visited on 09/30/2022).

[58] Unity Technologies. *Unity Documentation*. 2022. URL: https://docs.unity3d.com/Manual/index.html (visited on 10/02/2022).

[59] Blender Online Community. *Blender - Documentation*. Blender Foundation. Stichting Blender Foundation, Amsterdam, 2022. URL: https://docs.blender.org/ (visited on 10/02/2022).

Repositories and Databases

[60] Leonard Lessin/Science Source Stock Photos. *Image SS2836731 / 3C7166*. URL: https://www.sciencesource.com/ (visited on 09/30/2022).

[61] Benjamin Kahl. *RRad Project and Repository*. 2020. URL: https://github.com/Helliaca/RRad (visited on 09/30/2022).

[62] Simon Kallweit et al. *The Falcor Rendering Framework*. Mar. 2022. URL: https://github.com/NVIDIAGameWorks/Falcor (visited on 09/30/2022).

[63] Benjamin Kahl. *RTRad Project and Repository*. URL: https://github.com/Helliaca/RTRad (visited on 10/02/2022).

[64] *The Stanford 3D Scanning Repository*. 2022. URL: http://graphics.stanford.edu/data/3Dscanrep/ (visited on 09/30/2022).

[65] Crytek. *Sponza Scene (CryEngine Asset Database)*. URL: https://www.cryengine.com/marketplace/product/crytek/sponza-sample-scene (visited on 09/30/2022).

[66] Valve Corp. *Steam Hardware Survey (July 2022)*. URL: https://store.steampowered.com/hwsurvey/videocard/ (visited on 07/10/2022).

Lectures and Slides

[67] *FU Berlin Computer Graphics Lectures (Summer semester)*. 2020.

[68] *UCDavis Computer Graphics Lectures*. 2012. URL: https://www.youtube.com/watch?v=gLfYTP4F23g (visited on 10/02/2022).

[69] Philipp Slusallek, Karol Myszkowski, and Gurprit Singh. *Realistic Image Synthesis - Radiosity (Lecture Slides)*. Uni-Saarland, 2020. URL: https://graphics.cg.uni-saarland.de/courses/ris-2020/slides/RIS-02-Radiosity-2020.pdf (visited on 10/03/2022).

[70] Charles A. Wüthrichs. *Computer Graphics 7 - Radiosity (Lecture Slides) Bauhaus-Universität Weimar*. URL: https://www.uni-weimar.de/fileadmin/user/fak/medien/professuren/Computer_Graphics/7-globillu-radiosMG17.pdf.

[71] Cyril Crassin. *Octree-Based Sparse Voxelization for Real-Time Global Illumination (Slides)*. 2012. URL: http://www.icare3d.org/research/GTC2012_Voxelization_public.pdf (visited on 09/30/2022).

[72] Mike Bailey. *GLSL Geometry Shaders*. 2019. URL: http://web.engr.oregonstate.edu/~mjb/cs519/Handouts/geometry_shaders.1pp.pdf (visited on 10/02/2022).

[73] Matthew Szudzik. *An Elegant Pairing Function (Slides)*. 2006. URL: http://szudzik.com/ElegantPairing.pdf (visited on 09/30/2022).

Other

[74] Scratchapixel. *3D Viewing: the Pinhole Camera Model*. URL: https://www.scratchapixel.com/lessons/3d-basic-rendering/3d-viewing-pinhole-camera/how-pinhole-camera-works-part-1 (visited on 10/12/2022).

[75] Joey de Vries. *LearnOpenGL*. 2014. URL: https://learnopengl.com/ (visited on 09/30/2022).

[76] Scratchapixel. *Ray Tracing: Rendering a Triangle*. URL: `https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/moller-trumbore-ray-triangle-intersection` (visited on 10/02/2022).

[77] Hugo Elias. *Radiosity*. URL: `https://www.jmeiners.com/Hugo-Elias-Radiosity/` (Preserved on a website by J. Meiners). (Visited on 09/30/2022).

[78] Ingo Radax. *Instant Radiosity for Real-Time Global Illumination*. Tech. rep. TR-186-2-08-15. Favoritenstrasse 9-11/E193-02, A-1040 Vienna, Austria: Institute of Computer Graphics and Algorithms, Vienna University of Technology, 2008. URL: `https://www.cg.tuwien.ac.at/research/publications/2008/radax-2008-ir/` (visited on 09/30/2022).

[79] Jeremy Appleyard and Scott Yokim. *Programming Tensor Cores in CUDA 9*. [Nvidia Technical Blog] URL: `https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/`. 2017. (Visited on 10/02/2022).

[80] Andrew Burnes. *NVIDIA RTX: List Of All Games, Engines And Applications Featuring GeForce RTX-Powered Technology*. 2021. URL: `https://www.nvidia.com/en-us/geforce/news/nvidia-rtx-games-engines-apps/` (visited on 09/30/2022).

[81] Advanced Micro Devices, Inc. *A Foundation for High Performing Graphics - AMD RDNA 2 Explained*. URL: `https://www.amd.com/system/files/documents/rdna2-explained-radeon-pro-W6000.pdf` (visited on 03/26/2023).

[82] Daqi Lin. *Using RTX to Accelerate Instant Radiosity*. Jan. 2019. URL: `https://dqlin.xyz/tech/2019/01/24/dxr-ir/` (visited on 10/02/2022).

[83] Chris Wyman. *A Gentle Introduction To DirectX Raytracing*. URL: `http://cwyman.org/code/dxrTutors/dxr_tutors.md.html` (visited on 10/02/2022).

[84] Martin Stich. *Introduction to NVIDIA RTX and DirectX Ray Tracing [Nvidia Technical Blog]*. 2018. URL: `https://developer.nvidia.com/blog/introduction-nvidia-rtx-directx-ray-tracing/` (visited on 10/02/2022).

[85] Masaya Takeshige. *The Basics of GPU Voxelization*. 2015. URL: `https://developer.nvidia.com/content/basics-gpu-voxelization` (visited on 10/02/2022).

[86] Kurt Zimmerman. *Developing the Rendering Equations*. 1998. URL: `https://www.cs.princeton.edu/courses/archive/fall10/cos526/papers/zimmerman98.pdf` (visited on 10/02/2022).

[87] Emmett Kilgariff et al. *NVIDIA Turing Architecture In-Depth*. [Nvidia Technical Blog] URL: `https://developer.nvidia.com/blog/nvidia-turing-architecture-in-depth/`. 2018. (Visited on 10/02/2022).