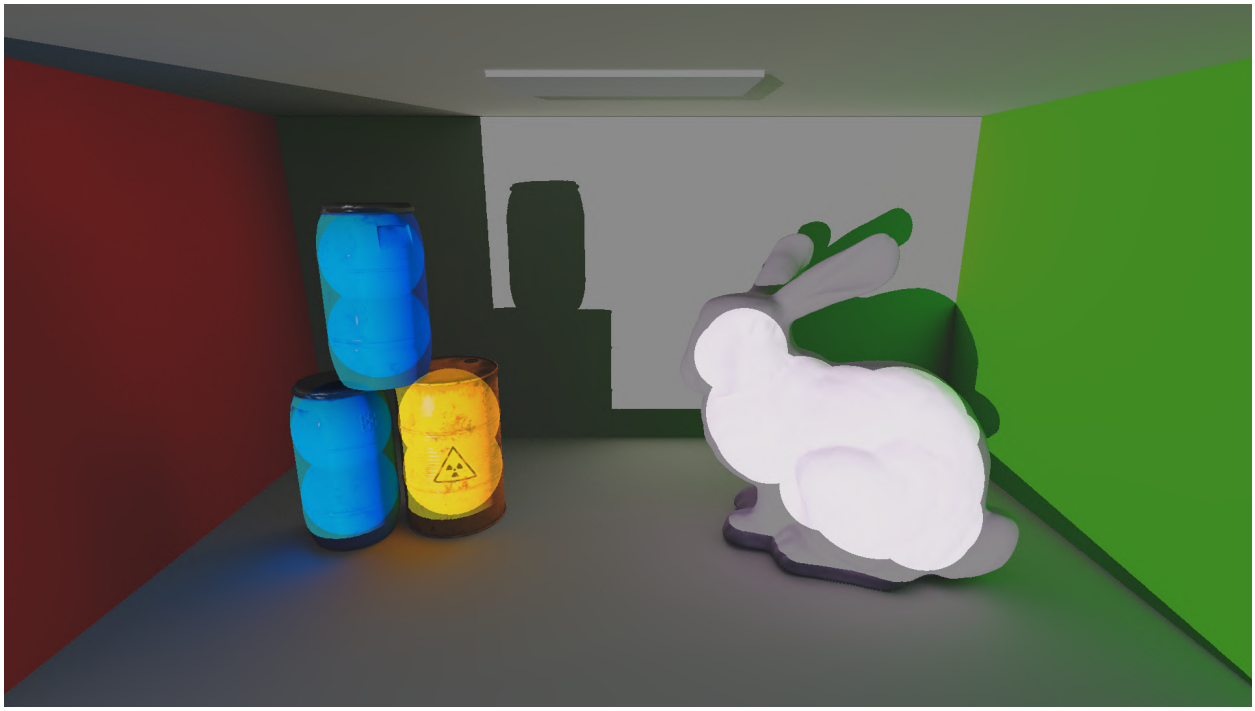

Evaluating the Use of Proxy Geometry for RTX-based Ray Traced Diffuse Global Illumination

Master's thesis in computer science and engineering

Simon Moos

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Master's thesis 2020

Evaluating the Use of Proxy Geometry for RTX-based Ray Traced Diffuse Global Illumination

Simon Moos

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

Evaluating the Use of Proxy Geometry for RTX-based Ray Traced Diffuse Global Illumination

Simon Moos

Supervisor: Erik Sintorn, Department of Computer Science and Engineering
Examiner: Ulf Assarsson, Department of Computer Science and Engineering
Advisor: Magnus Pettersson, RapidImages

Master's Thesis 2020
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Visualization of sphere set proxy geometry, with superimposed diffuse global illumination

Typeset in LaTeX
Gothenburg, Sweden 2020

Evaluating the Use of Proxy Geometry for RTX-based Ray Traced Diffuse Global Illumination

Simon Moos
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

Diffuse global illumination is vital for photorealistic rendering, but accurately evaluating it is computationally expensive and usually involves ray tracing. Ray tracing has often been considered prohibitively expensive for real-time rendering, but with the new RTX technology it can be done many times faster than what was previously possible. While ray tracing is now faster, ray traced diffuse global illumination is still relatively slow on GPUs, and we see the potential of improving performance on the application-level.

We examine how well proxy geometry can work for RTX-based, ray traced diffuse global illumination, in terms of rendering time and visual error. Three different types of proxy geometry are tested—simplified triangle meshes, sphere sets, and voxel planes—to evaluate whether it is possible to get faster rendering without introducing significant visual error. We also identify a few requirements that such proxy geometry should fulfill.

We find that it is possible to achieve faster rendering times with relatively small errors using proxy geometry. While all proxy types demonstrate different performance and error characteristics, for all evaluated scenes we find that there is a simplified triangle mesh proxy with lower errors than all other types, which is also faster to render than the reference. This cannot be said for any of the other proxy types.

Keywords: computer graphics, rendering, real-time ray tracing, global illumination, diffuse global illumination, proxy geometry, rtx.

Acknowledgements

First of all I would like to thank my supervisor Erik Sintorn, who has been a very valuable part of this thesis. I would also like to thank Ulf Assarsson, who has been the examiner of this thesis.

I would like to thank RapidImages for giving me access to their office and resources, and everyone there who have helped me with everything from IT support to answering my questions.

Finally I would also like to thank my friends and family who have supported me through this journey. In a project of this size you will experience many highs and many lows, and I'm very thankful for people who have been around for both.

Simon Moos, Gothenburg, July 2020

Contents

List of Figures

1

Introduction

1.1 Background

Computer graphics is the branch of computer science that deals with the generation and processing of digital images. It is today a vital part of many large industries, such as the film and games industries, but it is also used in less obvious areas such as medicine and commerce. All in all computer graphics has a large impact on many aspects of modern society.

Often, the goal is to generate photorealistic images, that is, images with accurately simulated real-world physics of electromagnetic radiation, or as we usually think of it: *light*. An 8 Watt light source—a common modern light bulb—emits around the order of 10^{19} photons per second[1], which illustrates the sheer complexity of this simulation. In practice, like with most simulations, an ensemble of algorithms is employed to solve approximations of the original problem.

1.1.1 Real-time rendering

Rendering is the branch of computer graphics that is concerned with the very last step of image generation: assigning colors to pixels. A common goal within rendering is to generate photorealistic images, and to do it in as short time as possible. In many situations, speed is not strictly critical but more of a convenience; for example when rendering images that will later be composited into video (offline rendering). However, in real-time rendering—where sequences of generated images should reflect user interaction as soon as possible—rendering speed is critical. There is no standardized time frame, but real-time usually implies that the generation of a single image should take no more than $1000/30 = 33.\overline{33}$ milliseconds, so that a frame rate of at least 30 frames per seconds (FPS) can be achieved.

The typical way of achieving real-time rendering is to use *rasterization*, which is a technique where images are drawn triangle-by-triangle and assigns color values to pixels that lie inside the triangle. Rasterization has many limitations when it comes to accurately simulating light, but is significantly faster than the alternative: *ray tracing*. Ray tracing more closely resembles how physical rays of light bounce around

[1] Assuming perfect efficiency (no heat generation) at a wavelength of 550 nanometers.

in real life, and adopts both the realism of accurate simulation and computational complexity that comes with that.

Global illumination (GI) is a phenomenon that is important to accurately capture for photorealistic rendering [Ake+18a, p. 438]. Global illumination is simply a name for light that reaches a camera or eye after bouncing/interacting with surfaces more than once. As can be seen in Figure 1.1, global illumination realistically lights up surfaces not directly hit by light and is clearly visually important. A cheap approximation such as ambient lighting can recreate some parts of the effect, but clearly fails to capture some of the intricacies of GI.

Accurate global illumination is difficult to achieve using rasterization, and for these situations other techniques have historically been employed (see [Ake+18a, chap. 11.5]). While these techniques work well in some cases they usually produce biased results, or require preprocessing or expensive update operations which limit how geometry, lights, and materials in a scene may change. Conversely, with ray tracing global illumination can be achieved quite easily, and ray tracing does not depend on precalculated light paths or visibility and therefore does not impose any strong limitations on scene changes. The ability to perform changes and observe their effects in real-time is a highly sought-after in games and visualization software.

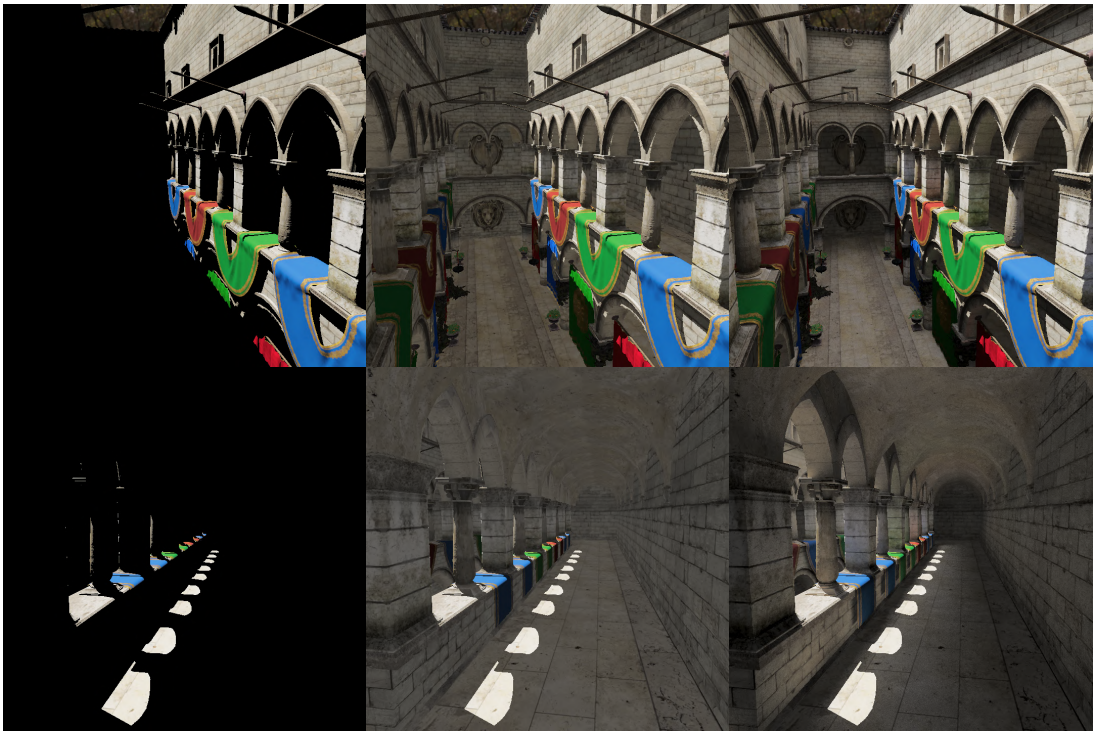

Figure 1.1: Impact of diffuse global illumination. Left: no global illumination. Middle: the *ambient* approximation, which applies a uniform and constant light across the whole scene. Right: path traced diffuse global illumination (with one light bounce).

The graphics processing unit (GPU) is a good candidate for performing real-time ray tracing in today's computers [Ake+18a, chap. 11.7], and recent technological ad-

vancements have made real-time ray tracing plausible on GPUs [Ake+18b; Maj+19; Sch+17], allowing for real-time ray traced global illumination. One of the most notable advancements is Nvidia's RTX technology [NVI18]. Nvidia claim that it is possible to get more than $10 \cdot 10^9$ rays per second (10 GRays/s) with the RTX technology [NVI18, p. 32]. Assuming a resolution of 1920×1080 (1080p) at 30 FPS with a single light bounce and a shadow ray per hit, $10 \cdot 10^9$ rays per second allows for approximately 40 samples per pixel (SPP)[2]. 40 SPP is far from enough for rendering noise-free images in general [Ake+18b, p. 8], and $10 \cdot 10^9$ rays per second is not achievable under more realistic cases; for example, see Figure 5.2. While promising, these techniques are clearly still very expensive and there are still limitations in what type of scene complexity can be captured. If real-time ray tracing could be made faster it would make ray traced global illumination suitable for more applications.

1.1.2 Proxy geometry

When rendering an object it is sometimes advantageous to use some geometry representation other than the original. For example, if the object is far away from the camera and barely visible it might be possible to use a lower fidelity geometry in its place without compromising the rendering quality significantly. Since this alternative geometry acts in place of some original it is often referred to as a *proxy geometry*. Rendering with this proxy, when applicable, can result in faster rendering since resources are not wasted on non-important (e.g., non-visible) details. In real-time rendering doing this can be critical in maintaining a good frame rate.

Depending on application, different types of geometry proxies might be more or less applicable. For example, if it is only relevant to know whether an object is visible or not (e.g., in occlusion culling [PT02]) a bounding box could suffice. On the other hand, when rendering shadows a bounding box would be a poor fit since the silhouette of the object is important; in such cases other types of proxy geometries have shown to be useful [Sil+14]. The most important property of a geometry proxy is that it captures the essential features of the geometry. As demonstrated with the above examples, however, what is essential is not universal, and proxies must be tailored for specific purposes.

1.1.3 Diffuse global illumination

This thesis will not focus on global illumination as a whole, but rather the subset often referred to as *diffuse* global illumination. From a physical perspective there is no distinction, since they describe the same phenomenon, but from the perspective of computer graphics it can be advantageous to keep the concepts separate due to their different performance characteristics and the fact that techniques for solving them often differ.

Diffuse global illumination effects emerge when light interacts with diffuse surfaces, such as a *Lambertian* surface. This type of surface is characterized by low coherency,

[2] $10 \cdot 10^9 / (4 \cdot 30 \cdot 1920 \cdot 1080) \approx 40.2$

where light scatters uniformly in all directions in the normal oriented hemisphere of the hit point. For the purpose of this thesis diffuse global illumination can be further split into two categories: large-scale and small-scale effects. Large scale effects includes color bleeding and ambient lighting, while on the small-scale we have what is often referred to as ambient occlusion (AO). The large-scale effects are particularly interesting in conjunction with proxy geometry, since it is a low-frequency effect, meaning that its value changes slowly across space, and that most high-frequency details of the geometry are lost.

1.2 Problem statement

The GPU is a good candidate for performing real-time ray tracing in today's computers, and with the advent of the RTX technology it is increasingly easy to accomplish. Proxy geometry is often used to speed up rendering, but there is not much research studying the potential performance benefits of using proxies with the RTX technology, nor what types of proxies are suitable.

This thesis will study existing and novel proxy geometry solutions to see if they can be used to speed up RTX-based ray traced diffuse global illumination. Formally, we wish to find proxy geometry that:

1. is compatible with the RTX APIs,
2. is faster to render ray traced diffuse global illumination with (using the RTX APIs and hardware), and
3. does not significantly decrease the quality of the rendered diffuse global illumination.

The latter two points are in comparison to the original/input geometry.

1.2.1 Limitations

There exist multiple material models which are diffuse in appearance and behaviour. For this thesis we will limit experiments to the Lambertian diffuse surface model.

1.2.2 Note on real-time ray tracing hardware & generality of the problem statement

At the time of writing this thesis real-time ray tracing is beginning to open up to non-vendor-specific APIs with Vulkan extensions such as `VK_KHR_ray_tracing`. With that said, only Nvidia are selling GPUs with ray-tracing hardware to consumers, so any attempt at non-vendor-specific discussion would inevitably be about Nvidia's proprietary technology. Therefore this thesis will be limited to the RTX APIs.

2

Previous work

This thesis is at its core about ray tracing, RTX ray tracing performance, and proxy geometry solutions. In this chapter we will highlight some previous research into these three subjects.

2.1 Ray tracing

A primitive version of ray tracing for image generation was first introduced by Arthur Appel in 1968. In 1980 Whitted introduced a version of ray tracing—today often referred to as *Whitted ray tracing*—which allows for effects such as reflections and refraction, and coined the term global illumination [Whi80].

Kajiya formulated the rendering equation in 1986 [Kaj86], which is a universal equation for describing the equilibrium between incoming, emitted, and outgoing light from a point on a surface. The equation is a variant of physically derived radiative heat transfer equations but in a form more suited for computer graphics. Compared to other contemporary models of light for computer graphics, such as the model used for Whitted ray tracing, the rendering equation is general and makes very few assumptions. Besides the equation an algorithm for solving it was also presented, which today is referred to as *path tracing*. Path tracing is an unbiased solution to the rendering equation, but is also very expensive in terms of rendering time.

Many different rendering algorithms based on ray tracing now exist, such as bidirectional path tracing and photon mapping, and it is a large and ongoing field of research. Since only path tracing will be used for this thesis we will not go into further details about other algorithms, and instead refer to the *Physically Based Rendering* book [PJH18].

2.2 Performance of GPU ray tracing and the RTX APIs

The performance characteristics of GPUs are well understood. Ray tracing on GPUs, however, is a newer concept and not as much research exists on the topic. Aila and Laine [AL09] try to discern performance bottlenecks in GPU ray tracing, specifically

on Nvidia GPUs. In the paper they identify problems with hardware work distribution which slow down ray tracing and suggest an algorithmic improvement that can be used to mitigate this problem. Additionally they note that the memory is not the bottleneck in most cases, but rather the previously mentioned work distribution. The only case they find where memory is the bottleneck is with diffuse rays.

In a more recent paper Lousada, Costa, and Pereira found that in general all GPU based ray tracing "benefits from a reduction in memory footprint and bandwidth" [LCP17]. As this work is also tested strictly on Nvidia GPUs, it suggests that improvements may have been made to Nvidia GPUs in accordance to the previous findings [AL09]. Similarly Ylitie, Karras, and Laine find that reducing the memory footprint of ray tracing data structures can significantly increase ray tracing performance on GPUs [YKL17].

The previously mentioned papers deal with software implementations of ray tracing written in general purpose programmable GPU hardware. The use of dedicated ray tracing hardware has been studied for a long time, for both partial solutions, e.g. dealing only with data structure construction [DFM13; Vii+18], and more complete solutions [Woo04; Spj+09; Nah+11; Nah+14]. Since this thesis will focus on the RTX-hardware we refer to Deng, Ni, Li, et al.'s survey on ray tracing hardware for more information [Den+17].

Nvidia first introduced their RTX-technology in a whitepaper [NVI18] about the capabilities of the new Turing GPU architecture. The whitepaper describes the *RT Cores*, which is the hardware that supports RTX ray tracing. Not much about the technology is revealed, but they acknowledge that a *bounding volume hierarchy* (BVH) is used internally in conjunction with hardware for intersecting rays with axis-aligned bounding boxes to speed up the BVH traversal [NVI18, pp. 30-31]. Also ray–triangle intersection testing is implemented in hardware [NVI18, p. 31].

Sanzharov, Gorbonosov, Frolov, et al. [San+19] perform black-box analysis on RTX-based ray tracing by rendering different scenes with different types of rays (glossy and diffuse) and recording rendering times. They note that RT cores seem to provide an improvement in incoherent memory access speeds, such as the workload of diffuse rays. While the relative speed of coherent to incoherent rays is lower with RT cores than previous GPU based implementations, diffuse is still significantly slower than more coherent rays.

2.3 Proxy geometry & generation algorithms

A triangle is the most basic polygon which spans an area in three dimensions (3D)—it is the 3D *simplex*—so all other polygons in 3D can be composed by a set of triangles. For this reason the triangle is the primitive used in most rasterizers, such as the one provided in GPUs, and triangles are often used in off-line and real-time rendering. While triangles are required when using the GPU rasterizer, a ray tracer is not as tied to using triangles as the input primitive.

For this thesis we will consider proxy geometry of not just triangles but all 3D

primitives. However, we will limit the search to algorithms that take triangle meshes as inputs, since it is the format that most assets are authored in.

A common algorithm for generating proxy geometry is *mesh simplification*. Mesh simplification is useful for level-of-detail (LOD) rendering, where an appropriate low-fidelity mesh is used depending on its distance from the camera. Since all the low-fidelity meshes must render similarly, it is important that some of their properties are kept (such as topology), and most mesh simplification algorithms provide some guarantees in this regard. Mesh simplification typically takes a triangle mesh as input and generates a similar triangle mesh with a smaller amount of triangles. These types of algorithms are thoroughly researched and there exist many variants with different advantages and constraints [Ake+18a, sec. 16.5][III04].

A mesh simplification algorithm typically defines an error metric which it tries to minimize while reducing triangles. Some mesh simplification algorithms have error metrics that encode geometrical constraints. Simplification envelopes and permission grids allow a user to define how the simplified mesh should be generated [III04]. Gaschler, Fischer, and Knoll describe a mesh simplification algorithm, *the bounding mesh algorithm* [GFK15], which guarantees that the simplified mesh bounds the input mesh.

Silvennoinen, Saransaari, Laine, et al. describe an algorithm for converting a triangle mesh to a *triangle soup*, i.e., a set of unorganized triangles without connectivity information [Sil+14]. The algorithm can generate triangle soups of very few triangles intended for use in occlusion tasks, such as shadows. However, it is not possible to retain color information and its triangles do not represent the surface of the input mesh. For these reasons it is not very suitable for shading or global illumination.

A *voxel* is a 3D primitive that can be used in rendering, often as part of a structure of many voxels, such as a grid. Gobbetti and Marton showed that voxels can be used as proxies for far-away triangles and through that decrease the total number of primitives considerably [GM05]. Representations that exclusively use voxels have also proven useful in several rendering tasks. Laine and Karras describe methods for generating and rendering voxels that represent large and complex geometry [LK10a]. Additionally they describe a method of carving cubic voxels by intersecting them with half-planes to more accurately approximate the silhouette of the input geometry (voxel contours) [LK10a; LK10b]. Voxels have also been proven useful as proxy geometry for global illumination rendering, for example in *voxel cone tracing* [Cra+11]. For all of the previously mentioned voxel-based techniques the data is stored in hierarchical data structures and the algorithms are tied into the traversal of the data structures. In RTX-based ray tracing the API does not allow for custom traversal logic, so none of these are directly applicable for this thesis.

A triangle mesh can be approximated by a set of arbitrary geometrical primitives, such as bounding boxes (e.g., see [PT02]). One useful primitive is the sphere, and there exist algorithms that approximate a triangle mesh as a set of spheres [BO02; Wan+06]. Bradshaw and O'Sullivan describe how to generate a set of spheres (optionally in a tree structure) and how the representation can been used for approxi-

mate collision detection and distance queries [BO02]. Ren, Wang, Snyder, et al. use another algorithm [Wan+06] to generate the same representation for use as proxy geometry for analytical occlusion calculations for soft shadows [Ren+06].

The *ray marching* algorithm [HSK89] is a variation on ray tracing, where instead of performing explicit ray–primitive intersection tests, discrete steps along a ray are evaluated. Exactly what is evaluated is not fixed, but it is common to use a *signed distance field* (SDF) geometry representation. An SDF represents an encoded surface, and in 3D is a function $D : \mathbb{R}^3 \to \mathbb{R}$, which returns the signed distance from the surface it encodes to the given point. Surfaces can be encoded using formulas [HSK89], but any arbitrary surface, such as the one defined explicitly by a triangle mesh, can also be encoded. For these cases a lookup table can be used in place of D.

3

Rendering theory

3.1 Measuring light

To reason about light, it is important to have a theoretical framework for it. As visible light is electromagnetic radiation within a range of wavelengths—approximately 380–780 nanometers [PJH18, ch. 5]—we often use the fields of physics knows as *radiometry* and *photometry*. Both fields are concerned with electromagnetic radiation and are mostly analogous to each other; the difference is that radiometry deals with absolute units, while photometry deals with the radiation as perceived by the human eye [Ake+18a, sec. 8.1]. For rendering it is common to use perceived *color*, e.g. defined in RGB, instead of a spectrum of light. This simplification allows us to ignore the photometric aspects of light in calculations, as it is already encoded into the concept of color [Ake+18a, sec. 8.1.4]. For that reason the units and theory presented in this section all come from radiometry.

Power is a measurement of the amount of light flowing through a point over a fixed amount of time, and is often measured in watt (**W**, joules per second) [McG19a]. For example, a light bulb of 100 **W** has a total of 100 joules of light energy leaving the bulb every second (assuming perfect efficiency).

For rendering, it is of interest to understand how light interacts with surfaces. To measure the amount of light arriving on a patch of surface, from all possible directions, *irradiance* is used. Irradiance is defined as watt per area ($\mathbf{W}/\mathbf{m}^2$) [McG19a].

Finally, it is useful to consider light along a ray—analogous to a physical beam of light—within the context of light–surface interactions. To measure the amount of light arriving or leaving a patch of surface in a single direction *radiance* is used. Radiance is defined as Watt per area per solid angle ($\mathbf{W}/(\mathbf{m}^2 \cdot \mathbf{sr})$) [McG19a]. The directionality is encoded using the unit *steradian* (**sr**) which is the three-dimensional sphere equivalent of a radian.

This is a very brief introduction to radiometric units, but should be sufficient for understanding this thesis. For more information we refer to *The Graphics Codex* [McG19b].

3.2 The rendering equation & global illumination

The rendering equation [Kaj86] describes how outgoing radiance (L_o) in direction ω_o from point $\mathbf{x}$ on a surface relates to emitted (L_e) and incoming (L_i) radiance:

$$L_o(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + \int_\Omega f_r(\mathbf{x}, \omega_i, \omega_o) L_i(\mathbf{x}, \omega_i)(\omega_i \cdot \mathbf{n}) \, \mathrm{d}\omega_i \tag{3.1}$$

where $\int_\Omega \mathrm{d}\omega_i$ is the integral over all directions ω_i in the normal-oriented hemisphere Ω, f_r is the BRDF which encodes how light scatters and is absorbed at $\mathbf{x}$, and $(\omega_i \cdot \mathbf{n})$ is the dot product of the incoming light direction and the surface normal $\mathbf{n}$. The BRDF is dependent on material, but some properties must hold, namely: $\int_{-\infty}^{\infty} f_r = 1$ and $f_r \geq 0$ so the laws of energy preservation hold. $L_i(\mathbf{x}, \omega_i)$ is identical to $L_o(\mathbf{x}', \omega_o')$ for some other surface point $\mathbf{x}'$ visible from $\mathbf{x}$ and $\omega_o' = \frac{\mathbf{x}-\mathbf{x}'}{\|\mathbf{x}-\mathbf{x}'\|}$; the rendering equation is therefore infinitely recursive. For an arbitrary scene it is not possible to analytically solve the equation [PJH18, ch. 1.2], so it must be solved through other means, such as simulation.

What the equation describes, in simplified terms, is that light leaving a surface is either emitted (the surface is a source of visible light) or scattered from other surfaces. The phenomenon that is referred to as global illumination is effectively the integral-part of the equation, where light is scattered resulting in inter-surface light interactions.

For Lambertian diffuse materials the BRDF is $\frac{\mathbf{c_x}}{\pi}$, where $\mathbf{c_x}$ is the reflectivity of the surface at point $\mathbf{x}$ [McG19c]. If all materials are assumed to have this BRDF the equation can be simplified to

$$L_o(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + \frac{\mathbf{c_x}}{\pi} \int_\Omega L_i(\mathbf{x}, \omega_i)(\omega_i \cdot \mathbf{n}) \, \mathrm{d}\omega_i \tag{3.2}$$

For the purpose of the Lambertian diffuse GI that this thesis deals with exclusively, this form is sufficient.

3.3 Ray tracing & path tracing

There exists no closed form solution to (3.1) but it is possible to iteratively solve it using Monte Carlo simulation. One unbiased solution to the equation is path tracing, which was presented together with the rendering equation as a reference solution [Kaj86]. To render a single pixel of an image using path tracing, a ray is first traced from the camera center through the pixel on the image plane to find $\mathbf{x}$ in the scene (see Figure 3.1). Calculating $L_o(\mathbf{x}, \omega_o)$ (3.3), where ω_o is the direction from $\mathbf{x}$ to the pixel, will then resolve a single iteration—or *path*—of path tracing:

$$L_o(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + f_r(\mathbf{x}, \mathbf{X}_\Omega, \omega_o) L_i(\mathbf{x}, \mathbf{X}_\Omega)(\mathbf{X}_\Omega \cdot \mathbf{n}) \tag{3.3}$$

where $\mathbf{X}_\Omega$ is a uniformly sampled direction in the normal oriented hemisphere (Ω). The average of n iterations of this will be equivalent to (3.1) as $n \longrightarrow \infty$. Since the equation is still infinitely recursive it is not possible to evaluate this either; however, there is a diminishing return for increasingly recursive depths[1], so in practice the recursion can be aborted at a fixed depth or when returns are insignificantly small.

Convergence speed (in terms of number of iterations) of (3.3) can be significantly improved by counting contribution from light sources separately with a shadow ray per hit. The use of a shadow ray is effectively just importance sampling the rendering equation with respect to the light sources, so there is no loss of accuracy or generality. However, the details on doing this for any number of arbitrary light sources is complex and beyond the scope of this thesis; instead we will describe the process for a single punctual light source, which is enough to reproduce the results of this work.

$L_o^*(\mathbf{x}, \omega_o)$ is the updated version of (3.3) which explicitly evaluates the light contribution from a punctual light source at position $\mathbf{x}^*$:

$$\begin{aligned} L_o^*(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + f_r(\mathbf{x}, \mathbf{X}_\Omega, \omega_o) L_i^*(\mathbf{x}, \mathbf{X}_\Omega)(\mathbf{X}_\Omega \cdot \mathbf{n}) \\ + \frac{f_r(\mathbf{x}, \omega^*, \omega_o) V(\mathbf{x}, \mathbf{x}^*) L_e(\mathbf{x}^*, -\omega^*)(\omega^* \cdot \mathbf{n})}{\|\mathbf{x}^* - \mathbf{x}\|^2} \end{aligned} \tag{3.4}$$

where $\omega^* = \frac{\mathbf{x}^* - \mathbf{x}}{\|\mathbf{x}^* - \mathbf{x}\|}$, and $V(\mathbf{x}, \mathbf{x}^*) = 1$ if there is visibility (i.e., no other occluding surface) between $\mathbf{x}^*$ and $\mathbf{x}$, and $V(\mathbf{x}, \mathbf{x}^*) = 0$ otherwise. The same relation between L_i^* and L_o^* holds as for L_i and L_o, as discussed above. Figure 3.1 contains a visualization of a single evaluation of (3.4), with a maximum depth of 2.

The technique for resolving visibility and next recursive hit point $\mathbf{x}'$ for L_i and V in path tracing is *ray tracing*. A single ray can be geometrically defined as an origin and distance along a direction from origin, in what we will refer to as the ray equation:

$$\mathbf{r}(t) = \mathbf{r}_o + t \cdot \mathbf{r}_d \tag{3.5}$$

For solving $L_i(\mathbf{x}, \omega_i)$, as part of path tracing, the ray $\mathbf{r}(t) := \mathbf{x} + t \cdot \omega_i$ is used to find next hit point $\mathbf{x}'$. Specifically, $\mathbf{x}'$ can be calculated as the closest intersection point of $\mathbf{r}(t)$ and geometric primitives p (e.g., triangles) in a scene S. A naive method of achieving this is to linearly iterate each $p \in S$ and test against $\mathbf{r}(t)$, but it can be significantly sped up with the use of an *spatial acceleration structure*. There exist many different types of spatial acceleration structures, but since for this work we will rely on built-in acceleration structures (see Section 3.4) we will not discuss this further. For more information on spatial acceleration structures we instead refer to the *Physically Based Rendering* book [PJH18].

[1]Remember $\int_{-\infty}^{\infty} f_r = 1$ and $(\mathbf{X}_\Omega \cdot \mathbf{n}) \leq 1$, and defining $a := f_r \cdot (\mathbf{X}_\Omega \cdot \mathbf{n}) \leq 1$ we can roughly express the equation as $a^d \cdot dL_i$, where d is the recursive depth, and ignoring the emittance term. Unless $a = 1$ for all evaluations (unlikely), $a^d = 0$ as $d \longrightarrow \infty$.

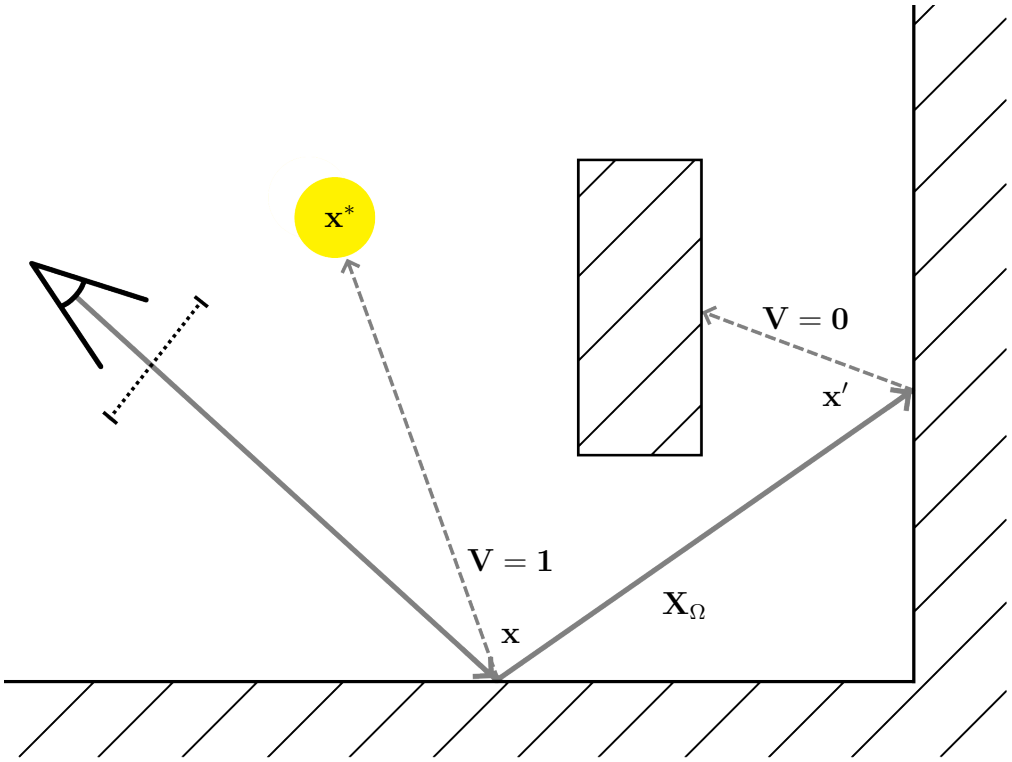

Figure 3.1: A single iteration (path) of path tracing with a maximum depth of 2, using the notation of (3.4).

3.4 The RTX APIs

In 2018 Nvidia released their proprietary RTX technology. RTX uses custom hardware, known as *RT cores* [NVI18], to speed up ray tracing. The technology is accessible through a set of APIs, hereinafter referred to as the *RTX APIs*. These include OptiX, DirectX Raytracing (DXR), and Vulkan through the `VK_NV_ray_tracing` extension.

The RTX APIs take control of acceleration structure building and traversal. Since implementations exist in proprietary driver code it is not known exactly what techniques are being used for these tasks, but from Nvidia's whitepaper [NVI18] it is known that some type of Bounding Volume Hierarchy (BVH) is used as a spatial acceleration structure. The hardware supporting the RTX ray tracing on compatible GPUs, the RT core, includes ray–box and ray–triangle intersection testing [NVI18]. The ray–box test supports the BVH traversal, while the ray–triangle test is available for speeding up triangle-based geometry.

The APIs provide a few core concepts for setting up ray tracing with arbitrary geometry. While the concepts are common to all RTX APIs we will use `VK_NV_ray_tracing` [Khrb] and `GLSL_NV_ray_tracing` [Khra] terminology here to be able to provide concrete examples and names.

Intersection geometry is constructed using `VkGeometryNV` instances, which contain

buffers for either triangles or axis-aligned bounding boxes (AABBs) and associated metadata. Shading is performed using a closest-hit shader, and optionally also an any-hit shader. Together these form what is known as a *hit-group*. When using triangle data the built-in ray-triangle intersection test is used, but when using AABB data a custom intersection shader is also supplied, as part of the hit-group. The intersection shader is responsible for reporting ray–geometry intersections for the custom geometry bounded by the AABB. Since custom intersection shaders run on the programmable parts of the GPU, some performance overhead is to be expected, compared to the ray–triangle intersection which is fully fixed.

The previously mentioned BVH acceleration structure is composed of two levels of separate BVHs: the top-level acceleration structure (TLAS) and the bottom-level acceleration structure (BLAS). A BLAS is constructed from one or more `VkGeometryNV` objects. A TLAS composes one or more BLAS, by associating a BLAS with an arbitrary transformation. The same BLAS can be referred to many times within the same TLAS with different transformations.

4

Generating & rendering proxy geometry

Previous chapters have discussed GPU performance and the theoretical reasons why using proxy geometry could be beneficial for different rendering tasks. In this chapter we will describe the three proxy geometry types that have been evaluated for this thesis. Both the generation and rendering of them will be discussed, together with motivations for why they were chosen.

4.1 Requirements for proxy geometry

There exist many different types of proxy geometry, as highlighted in Section 2.3, and not all of them are suitable for all use cases. To help evaluate whether a proxy may be suitable for use in ray traced diffuse global illumination, we have developed a set of requirements that should optimally be fulfilled. A suitable proxy should:

1. provide sufficient surface normals $\mathbf{n}$ for shading and recursive rays,
2. provide sufficient color information $\mathbf{c_x}$ for shading, and
3. be fully bounded by the source geometry.

The importance of Item 1 and Item 2 comes from the fact that $\mathbf{c_x}$ and $\mathbf{n}$ both occur in the diffuse rendering equation (3.2), so their inclusion is vital for the diffuse global illumination calculations.

The importance of Item 3 is not as obvious, but rendering with proxy geometry not fulfilling the requirement results in artifacts, as can be seen in Figure 4.1. While it manifests as two different types of artifacts (on the surfaces of the model itself, and on other surfaces) the reason is the same for both: rays hitting the proxy geometry where it is not expected.

If Item 3 is fulfilled for a proxy, its silhouette will cover a smaller or equal area to that of the original geometry, from all angles. This means that light leakage artifacts are possible. This type of artifact is of the same nature as the artifacts the requirement tries to resolve, except inversely so: a lack of darkening (ambient occlusion) where it should be dark. However, we argue that this artifact is less visible

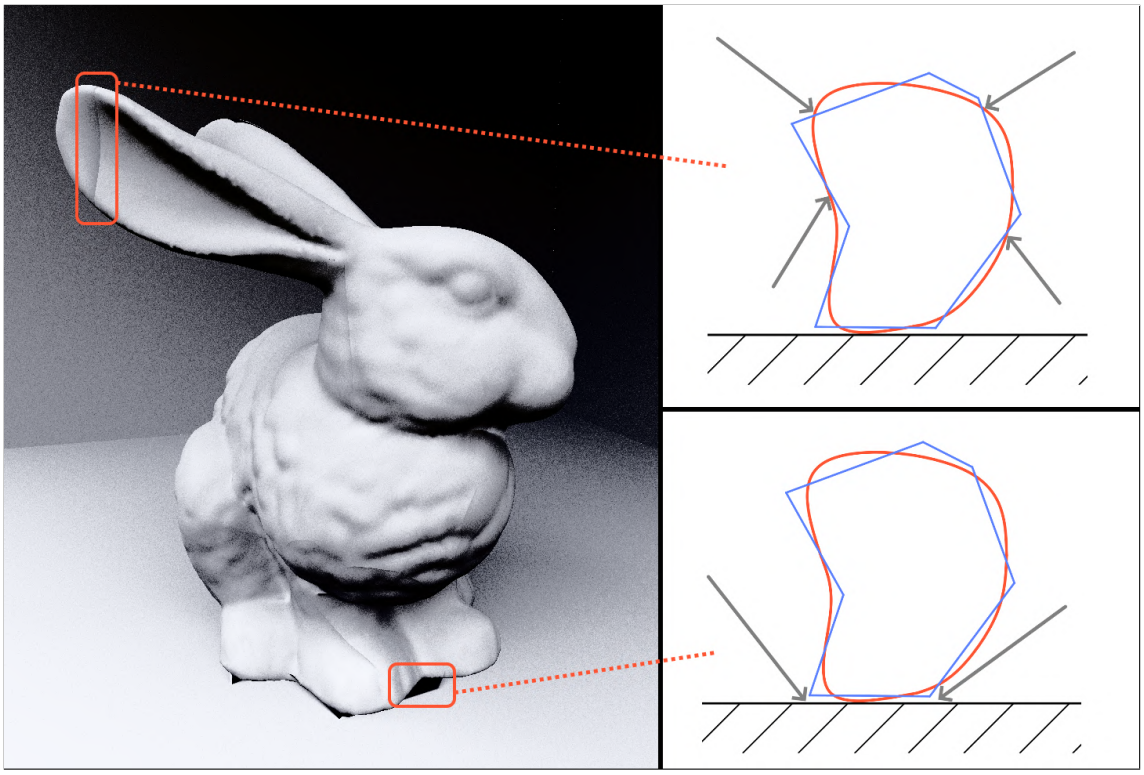

Figure 4.1: Artifacts from not fulfilling the bounding requirement. Left: Rendering using a non-bounding proxy (contrast increased so artifacts become more visible). Right: 2D schematic illustrating the source of the artifacts. The red curve represents the surface of the original geometry and the blue represents the proxy geometry. The gray arrows are examples of rays which, after diffusely scattering, would interact with the bad proxy geometry and cause pixels with the artifacts shown on left.

to the eye, comparing Figure 4.1 with the similar figures in Chapter 5 (results), which uses strictly conforming proxies. Additionally, a lack of darkening can in theory be mitigated by complementing the indirect light with a separate ambient occlusion (AO) system. How well a mitigation like this could work in practice is evaluated in Section 5.4.5.

4.2 Proxy geometry generation algorithms

To evaluate the question of this thesis, a few different types of geometry proxy will need to be evaluated. Three types of proxies have been chosen for this work and they will be discussed in detail in the remainder of this chapter.

For simplicity it is assumed that a triangle mesh input is watertight, meaning there are no gaps or openings in the mesh. This is important for some proxy geometry generation algorithms, where the distinction between the inside and outside of a mesh is required.

4.2.1 Simplified triangle meshes

As the RTX hardware has dedicated circuitry for ray–triangle intersection, triangle based proxies should in theory have an advantage over other types of proxies. Many triangle based proxies exist, but only a few could be considered feasible in accordance with the requirements established in Section 4.1. For example, the planar occluder sections proxy [Sil+14] will generate planes with normals that do not resemble those of the original geometry. Traditional mesh simplification—for example often used for level-of-depth (LOD)—is interesting since it attempts to maintain most of the properties of the original mesh geometry, just at a lower resolution. A mesh simplified in such ways will have normals similar to the normals of the original mesh, and color can be encoded the same way, for example using texture mapping. However, special care is needed to guarantee the bounding requirement (Item 3 Section 4.1).

Mesh simplification is a widely researched topic, so for the sake of scope we will not make any attempt at creating a novel algorithm; instead we rely on existing algorithms and implementations. Of the many existing mesh simplification algorithms, only a few could be identified that have mechanisms for controlling the bounds of the simplified mesh. Simplification envelopes and permission grids (as discussed in [III04]) are interesting, but no publicly available implementation of either can be found. The more recent *bounding mesh algorithm* [GFK15], however, has an open-source implementation[1] which was used for all mesh simplification in this work. As its name implies, the main goal of the bounding mesh algorithm is to create a mesh bounding some other mesh. However, the algorithm (and implementation) supports inverting a constraint to give the opposite result—what they refer to as an inner bounding mesh. In this text we are only interested in inner bounding meshes, or bounded meshes, so all further description of this algorithm will use the inverted constraint.

The bounding mesh algorithm simplifies a mesh by iteratively performing *edge contractions*, one by one, until some break condition occurs, e.g., the number of triangles is below a predefined limit. An edge contraction takes an edge $e = (\boldsymbol{v}_1, \boldsymbol{v}_2)$ and a new position $\boldsymbol{v}$, and merges the two vertices $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ into a new vertex at position $\boldsymbol{v}$. Any neighbouring triangle that has become degenerate (zero area) is also removed.

Since an edge contraction is a destructive operation, the order of execution matters. For each edge e in the mesh, an optimal $\boldsymbol{v}^*$ is calculated by minimizing an error E, and the edge is placed into a priority queue sorted by error. This allows the edge contraction with the smallest errors to be performed first. The optimal position $\boldsymbol{v}^*$ and the error E are calculated as:

$$\min_{\boldsymbol{v}^*} E\left((\boldsymbol{v}_1, \boldsymbol{v}_2), \boldsymbol{v}^*\right) \text{ s.t. } \forall \boldsymbol{p} \in P(\boldsymbol{v}_1) \cup P(\boldsymbol{v}_2) : \boldsymbol{p}^{\mathrm{T}} \boldsymbol{v}^* \leq 0 \tag{4.1}$$

$$E(e, \boldsymbol{v}) = \sum_{\boldsymbol{p} \in P(e)} d(\boldsymbol{p}, \boldsymbol{v})^2, \tag{4.2}$$

[1] The implementation can be found at https://github.com/gaschler/bounding-mesh.

where P denotes the neighboring planes of the vertex or edge, and $d(\boldsymbol{p}, \boldsymbol{v})$ is the Hausdorff distance—the maximum distance—between the two arguments. Minimizing the Hausdorff distances ensures that the simplified mesh is as similar to the original mesh as possible, while the $\boldsymbol{p}^{\mathrm{T}}\boldsymbol{v}^* \leq 0$ constraint ensures that the simplified mesh is bounded within the original mesh [GFK15].

4.2.1.1 Algorithm

The bounding mesh algorithm can now be described:

1. for every edge e, find $\boldsymbol{v}^*$ and add $E(e, \boldsymbol{v}^*)$ to a priority queue
2. while no break conditions are active, repeat:
   (a) pop the smallest $E(e, \boldsymbol{v}^*)$ from the priority queue
   (b) perform the edge contraction $e \longrightarrow \boldsymbol{v}^*$
   (c) for every edge e' now connected to $\boldsymbol{v}^*$, find a new optimal edge contraction point $\boldsymbol{v'}^*$ and update the error in the priority queue for e' to $E(e', \boldsymbol{v'}^*)$

4.2.1.2 Encoding color

Triangles allow for many ways of encoding color information. Two common approaches are vertex colors and texture mapping. While vertex colors have a smaller memory-footprint in most realistic cases, texture mapping has a few advantages:

- assuming identical parameterization, textures can be swapped to change materials,
- the resolution of textures can be adjusted to balance the accuracy–memory-footprint trade-off,
- texture mapping is commonly used for triangle meshes, and all input triangle mesh geometry uses texture mapping.

For these reasons texture mapping was used for all simplified triangle meshes. However, note that the publicly available implementation of the bounding mesh algorithm does not support texture mapping, which means that some additional mesh processing is required for this (see Section 5.2 for more information).

4.2.2 Sphere set

The sphere is an interesting primitive to test for this thesis since it inherently encodes a significant amount of information with very few parameters. A single sphere can be parameterized as a center point and a radius, and describes a volume visible from all possible directions and an infinite amount of surface normals. Compare this to a triangle which requires three points, encodes a single normal, and does not have a volume[2].

[2]For this purpose a triangle could also be considered an infinitely thin volume, if both possible normals are accepted as valid.

It has been shown that a triangle mesh can be approximated by a set of spheres [BO02; Wan+06; Ren+06]. However, due to the bounding requirement (Item 3 Section 4.1) none of those techniques can be applied directly; in fact, they all have the inverse requirement of bounding the original mesh. As Wang, Zhou, Snyder, et al. [Wan+06] claim to achieve better results than previous techniques (i.e., [BO02]), we will base our problem formulation on the former.

The optimization problem of "Variational Sphere set Approximation for Solid Objects" [Wan+06] is formulated as:

$$\begin{aligned} &\min \sum_{i=1}^{n_s} V\left(T, S_i\right) \\ &\text{s.t. } V(T, S_i) \geq 0, \forall S_i, \end{aligned} \tag{4.3}$$

where T is the triangle mesh, S_i is sphere i in a set of n_s spheres, and $V(T, S_i)$ computes the signed volume inside the sphere S_i but outside the triangle mesh T. Solving this will give an optimally tight, but bounding, fit of the triangle mesh. The inverse problem can be formulated as:

$$\begin{aligned} &\max V\left(\bigcup_{i=1}^{n_s} S_i\right) \\ &\text{s.t. } V(T, S_i) \leq 0, \forall S_i, \end{aligned} \tag{4.4}$$

where the objective function is the volume of the union of all spheres. A solution to this problem for a given n_s will give a set of spheres that is bounded by the triangle mesh while optimally filling the interior of the mesh volume. This solution fulfills the requirement while maximizing the total volume union, and hopefully also retains as much of the object silhouette as possible.

While a closed form solution of the objective function—i.e., the volume of the union of a set of spheres—does exist [ABI88], it is a complex calculation to perform, and it can be rewritten as follows:

$$\begin{aligned} V\left(\bigcup_{i=1}^{n_s} S_i\right) &= \sum_{i=1}^{n_s} V(S_i) - V\left(\bigcap_{i=1}^{n_s} S_i\right) \\ &= \sum_{i=1}^{n_s} V(S_i) - \alpha \sum_{i=1}^{n_s} \sum_{j=i+1}^{n_s} V\left(S_i \cap S_j\right), \end{aligned} \tag{4.5}$$

for $0 < \alpha \leq 1$. No closed form of the volume of the intersection of a set of n spheres is known to exist [CS15], but it is simple for $n = 2$ (4.6). Adding all pairwise intersection volumes will overestimate the total intersection volume for all cases

where more than two spheres intersect; however, this can be mitigated by choosing a good α (on a case-by-case basis). The intersection volume of two spheres can be calculated as:

$$\begin{aligned} & V\left(S_i \cap S_j\right) = V\left(r_i, r_j, d\right) = \\ & = \tfrac{4\pi r_i^3}{3}, && d < r_j - r_i \\ & = \tfrac{4\pi r_j^3}{3}, && d < r_i - r_j \\ & = \tfrac{\pi(r_i + r_j - d)^2\left((r_i + r_j + d)^2 - 4\left(r_i^2 - r_i r_j + r_j^2\right)\right)}{12d} && |r_i - r_j| \leq d \leq r_i + r_j \\ & = 0 && d > r_i + r_j, \end{aligned} \tag{4.6}$$

where r_i and r_j are the radii of the spheres S_i and S_j, respectively, and d is the distance between the sphere centers [CS15].

The optimization problem used to generate a bounded sphere set solution is thus:

$$\begin{aligned} \max \ & \sum_{i=1}^{n_s} V(S_i) - \alpha \sum_{i=1}^{n_s} \sum_{j=i+1}^{n_s} V\left(S_i \cap S_j\right) \\ \text{s.t. } & V(T, S_i) \leq 0, \forall S_i. \end{aligned} \tag{4.7}$$

4.2.2.1 Algorithm

To solve (4.7) an algorithm similar to the presented algorithm for (4.3) [Wan+06] is employed:

1. uniformly select n_s random sphere centers from inside T
2. maximally increase the radius of each sphere while making sure the sphere does not intersect any triangle $t \in T$
3. for each sphere, perform local optimization on center and radius
   (a) if the optimization did increase the objective function, repeat
   (b) if the optimization failed to increase the objective function, first perform sphere teleportation then try optimizing again. If teleportation & optimization repeatedly failed to increase the objective function, assume an optimal solution has been found and abort.

To uniformly sample points inside the mesh a voxel representation of the mesh was utilized. Each triangle was voxelized to create a voxel-"shell" of the mesh, which would then be filled. To sample a random position inside the mesh a random inside-voxel (i.e., not part of the shell) was sampled and its center point was used as the sphere center location.

The local optimization step uses constrained black-box optimization, specifically Powell's BOBYQA algorithm [Pow09], to optimize the objective function. To dis-

courage spheres from intersecting T, a large penalty is applied to the objective function for that case. To guarantee that spheres are bounded by T, the optimization is constrained to never move a sphere center further than its current radius.

Sphere teleportation as a concept is borrowed from "Variational Sphere set Approximation for Solid Objects" [Wan+06] and is used to escape local minima. All spheres in the set are sorted relative to their volumes, and the s_n/k smallest are relocated and expanded according to step 1 and 2. k can be adjusted to weigh toward more aggressive (small k) or potentially less potent (large k). In practice a value of $k = 8$ works in most cases. The teleportation is admittedly noisy and heuristic, but multiple other teleportation strategies were tested and no better one could be found.

4.2.2.2 Encoding color

There are multiple viable color encodings for a sphere set. Considering a cube with a unique color per face with a sphere set proxy of a single sphere, it is clear that having a single color is not a viable encoding. A sphere must be able to encode different colors for different surface points.

We opted for a spherical harmonic (SH) color encoding, since it offers a smooth encoding and an adjustable level of precision, or degree, ℓ. As ℓ increases, so does the precision of the encoding, but also the memory-footprint of it, so the trade-off is easily controlled. Spherical harmonics is a way of mapping arbitrary functions to a sphere, by adding one or more scaled basis functions [Gre03]. The degree indirectly specifies the number of basis functions, which in turn decides the precision of the mapping. Spherical harmonics of small degrees are good at mapping low-frequency functions, such as diffuse colors, since a SH is smooth and does not incur aliasing. Conversely, spherical harmonics are not good at capturing high-frequency functions, unless the degree is high. For all results presented in this work an ℓ-2 spherical harmonic color encoding was used. Figure 4.2 contains a visualization of the directional coefficients of spherical harmonics, up to degree ℓ-2.

Another viable option is texture mapping. A spherical mapping can be used to convert sphere points in $\mathbb{R}^3$ to texture coordinates in $\mathbb{R}^2$, and the precision–memory-footprint trade-off can be adjusted with texture resolution.

4.2.2.3 Testing for intersection

A point $\mathbf{x}$ is on a sphere with center point $\mathbf{c}$ and radius r iff $\|\mathbf{c} - \mathbf{x}\| - r = 0$. To find an intersection between this sphere and a ray we can insert the ray equation (3.5) in place of $\mathbf{x}$. Additionally the equation can be simplified by assuming $\mathbf{c} = \mathbf{0}$ and redefining the ray origin as $\mathbf{r}'_o \coloneqq \mathbf{r}_o - \mathbf{c}$. We then have to solve the equation:

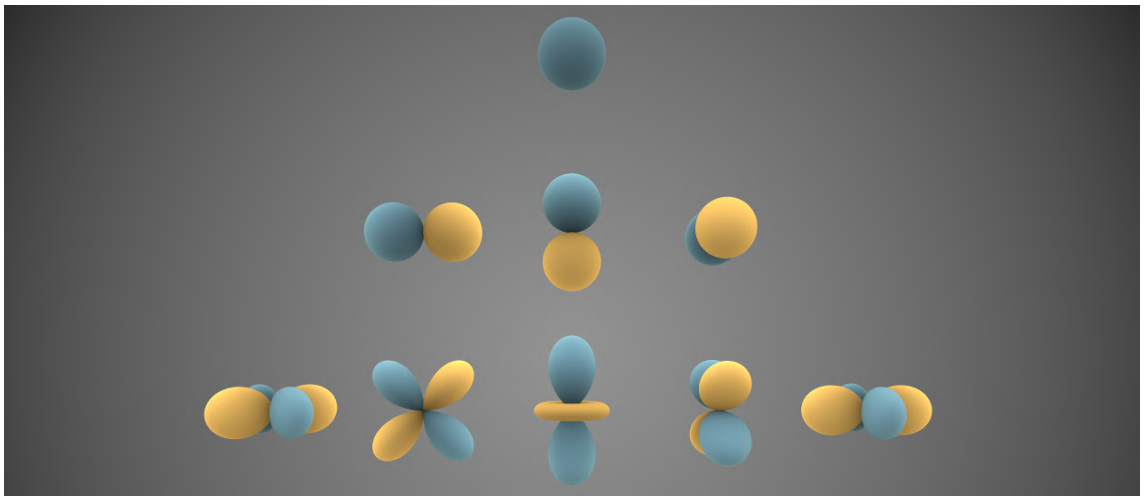

Figure 4.2: Visualization of the directional coefficients of spherical harmonics, up to degree ℓ-2. Blue represents positive function values and yellow negative function values. *Image courtesy of Inigo Quilez* [Qui13].

$$\begin{aligned} \|\mathbf{r}'_o + t\mathbf{r}_d\| - r &= 0 \\ \|\mathbf{r}'_o + t\mathbf{r}_d\|^2 - r^2 &= 0 \\ (\mathbf{r}'_o + t\mathbf{r}_d)^\mathsf{T}(\mathbf{r}'_o + t\mathbf{r}_d) - r^2 &= 0 \\ t^2 + t(2 \cdot \mathbf{r}_d^\mathsf{T}\mathbf{r}'_o) + \|\mathbf{r}'_o\|^2 - r^2 &= 0, \end{aligned}$$

which is a quadratic function where all real solutions correspond to geometrical intersection points. Given an intersection point $\mathbf{x}$, a surface normal can be calculated as:

$$\mathbf{n} = \frac{\mathbf{x} - \mathbf{c}}{\|\mathbf{x} - \mathbf{c}\|}.$$

4.2.3 Voxel planes

Voxels are interesting primitives to test for this thesis for multiple reasons. Foremost they are a very common geometrical representation in computer graphics and are also used in some global illumination algorithms (see Section 2.3). Additionally, voxels are easy to generate and the resolution—that is, the number of primitives to render—is trivially adjustable. Finally, they align closely with the geometry definition of the RTX APIs, as the APIs take axis-aligned bounding boxes for non-triangle intersection geometry.

However, voxels are not directly applicable for use with ray traced diffuse global illumination. A voxel only has six geometric normals, which is hard to fit arbitrary geometry to. Using voxel contours [LK10b] it is possible to achieve a better geometrical fit, by carving voxels with half-spaces while hierarchically traversing a tree

of voxels. In the RTX APIs there is no mechanism to interact with geometry at traversal, but carving voxels is still possible at the leaf-level. Since it is advantageous to keep the memory footprint low, we limited the number of half-spaces to one, effectively defining a plane inside the volume of a voxel: a *voxel plane*.

When using a single plane per voxel there is no guarantee that a grid of voxel planes can represent the full surface of the original geometry. For example, all concave meshes will produce voxel planes with *gaps* between them, as can be observed in Figure 4.3. In practice, strictly convex triangle mesh objects are exceptional, so these gaps will appear in voxel plane proxies for most input geometry.

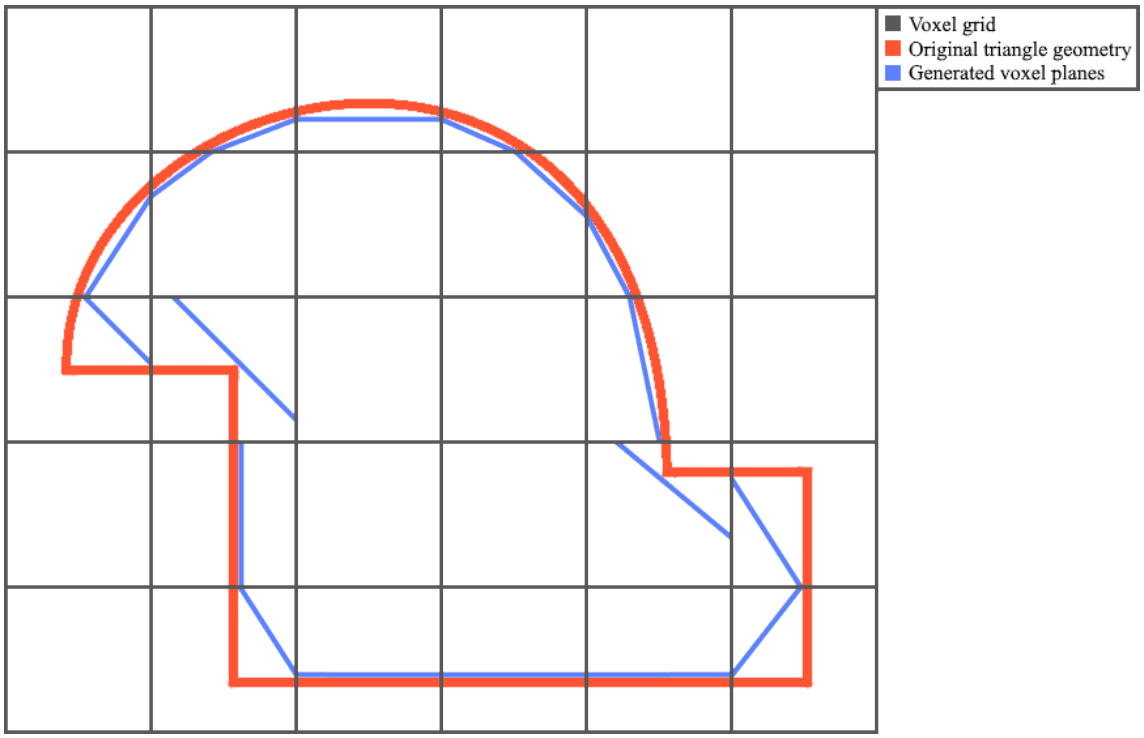

Figure 4.3: 2D example of a voxel plane proxy generated from a triangle mesh (assuming triangles are small). Note that sharp convex features, such as the outer corners, become less sharp, and that gaps appear between voxel planes at concave regions.

Since a voxel plane is one-sided and presumably quite small in relation to the size of the object it represents, it is reasonable to use a single uniform color per voxel plane.

4.2.3.1 Algorithm

A custom algorithm was developed to generate a set of voxel planes for a mesh. For a given grid dimension $\mathbf{D} \in \mathbb{N}^3$ and n_c, the number of quantized colors to use:

1. create a grid of resolution $\mathbf{D}$,
2. voxelize each triangle $t \in T$ and store the color of the intersecting triangle,

3. quantize the colors to n_c colors,
4. for each voxel, define a single plane of the same color as the voxel.

Each voxel plane is assigned the color of the intersecting triangle by projecting the voxel center point onto the triangle and sampling its texture at the interpolated texture coordinates for the projected point. If multiple triangles intersect a single voxel the color of the last triangle is used. The average color of all intersecting triangles could also be used for this.

The color quantization is done to decrease the memory footprint of colors. By approximating colors we go from a resolution-dependent number of colors to a constant n_c colors. The quantization is performed using k-means clustering with $k = n_c$.

Generating a plane for a voxel is done by finding a *consensus normal* $\tilde{\mathbf{n}}$ which is used for the plane, and adjusting the distance so the plane is fully behind all intersecting triangles. For a voxel with n intersecting triangles t_i with normals $\mathbf{n}_i$, $i = 1, 2, \ldots, n$:

$$\tilde{\mathbf{n}} = \frac{1}{n} \sum_{i=1}^{n} \mathbf{n}_i. \tag{4.8}$$

For $n = 1$ we have $\tilde{\mathbf{n}} = \mathbf{n}_1$ and for $n > 1$ it is the average normal. The intention is that $\tilde{\mathbf{n}}$ and the voxel plane represents the distribution of normals of the input geometry. For cases where there is much curvature within a single voxel, however, a single-plane representation can be poor. For example, consider a voxel with two intersecting triangles of almost perpendicular surface normals. The average of these normals will not necessarily represent the true distribution of normals. Increasing the resolution $\mathbf{D}$ can mitigate this problem, but at the cost of using more voxel planes.

To satisfy the bounding requirement (Item 3 Section 4.1) the plane with normal $\tilde{\mathbf{n}}$ must lie behind all intersecting triangles. To find the distance d of the plane along the normal from the origin we clip all intersecting triangles to the voxel bounds and find the smallest projected distance:

$$d = \min_{t_i} \min_{\mathbf{p} \in C(t_i)} \tilde{\mathbf{n}} \cdot \mathbf{p} - \epsilon, \tag{4.9}$$

where $C(t_i)$ is the set of triangle vertices clipped to the voxel bounds, and $\epsilon > 0$ and close to zero, to assure that the bounding requirement is fulfilled.

4.2.3.2 Testing for intersection

A voxel plane is defined by an axis-aligned bounding box (AABB) and plane. To test for intersection we first find the point $\mathbf{x}$ where the ray intersects the plane, and then test if $\mathbf{x}$ is inside the AABB.

A plane is defined by a normal and a distance, in this case $\tilde{\mathbf{n}}$ and d; a point $\mathbf{x}$ lies on the plane iff $\tilde{\mathbf{n}} \cdot \mathbf{x} + d = 0$. We insert the ray equation (3.5) and solve for t:

$$\begin{aligned}\widetilde{\mathbf{n}} \cdot (\mathbf{r}_o + t \cdot \mathbf{r}_d) + d &= 0 \\ \widetilde{\mathbf{n}} \cdot \mathbf{r}_o + t\widetilde{\mathbf{n}} \cdot \mathbf{r}_d + d &= 0 \\ t\widetilde{\mathbf{n}} \cdot \mathbf{r}_d &= -(d + \widetilde{\mathbf{n}} \cdot \mathbf{r}_o)\end{aligned}$$

Assuming $\widetilde{\mathbf{n}} \cdot \mathbf{r}_d \neq 0$

$$t = -\frac{d + \widetilde{\mathbf{n}} \cdot \mathbf{r}_o}{\widetilde{\mathbf{n}} \cdot \mathbf{r}_d}.$$

Finally, an AABB can be fully defined by two points, $\mathbf{b}_{\min} \prec \mathbf{b}_{\max}$, and a point $\mathbf{x}$ is contained inside the AABB iff $\mathbf{b}_{\min} \preccurlyeq \mathbf{x} \preccurlyeq \mathbf{b}_{\max}$ (where $\prec$ and $\preccurlyeq$ are component-wise vector comparisons).

5

Results

5.1 Evaluation methodology

Evaluating the thesis of this work necessitates implementing both a set of proxy geometry generation algorithms and a renderer that can render the original meshes and their proxies. Using these implementations the proxy geometries will be evaluated according to the criteria established in Section 1.2. In practice this means

1. measuring rendering times with and without proxies for direct comparisons, and
2. capturing images with and without proxies for visual comparisons.

To compare two images X and Y of identical dimensions $w \times h$ pixels we apply a per-pixel error metric for each pixel i:

$$E_i(X,Y) = \text{luminance}\left(|X_i^{(R)} - Y_i^{(R)}|, |X_i^{(G)} - Y_i^{(G)}|, |X_i^{(B)} - Y_i^{(B)}|\right), \tag{5.1}$$

where luminance is a function that returns the sRGB grayscale luminance [Ake+18a, p. 278] for a given sRGB color.

For most result images below, an error map will be presented alongside, where per-pixel values of $E_i(X,Y)$ are mapped to a color gradient (Figure 5.1). These images are meant to help visualize error across regions of the rendered images, for the benefit of the reader. Unless otherwise noted, all error maps calculate error in indirect-light-only images, which ignore all color except for bounce lighting. This means that features that absorb much light (e.g., dark and black colors) do not nullify the error in those regions.

Figure 5.1: The color gradient used for all error maps presented in this paper.

By default the renderer traces a shadow ray from each hit point of an indirect ray. This is often done in path tracing to speed up the convergence of the image and is described in Section 3.3. All images presented in this text are therefore rendered with

shadow rays. However, in all instances where rendering performance is presented in rays per second (specifically GRays/s for $\cdot 10^9$ rays per second), shadow rays are explicitly disabled when measuring. The purpose of this is to allow the ray per second metric to be strictly about diffuse rays, as it is the most relevant type of ray for this work. We believe this will give a more fair assessment of rendering times for different types of proxy geometries.

5.2 Implementation details

The renderer[1] was written using Vulkan and the `VK_NV_ray_tracing` extension was used to access the RTX ray tracing capabilities. The renderer is a hybrid renderer, meaning that is does both rasterization and ray tracing. The rasterizer renders first-hit visibility and direct light, and writes albedo, normals, and non-linear depth to a G-buffer. *Deferred ray tracing* [Ake+18b, p. 9] is used for indirect rays, meaning that secondary rays are generated from the G-buffer as described by Barré-Brisebois, Halén, Wihlidal, et al. [Bar+19]. Original triangle meshes are always used for the rasterization, while proxy geometry is strictly used for the indirect rays, including indirect shadow rays. The renderer uses a maximum depth of 1 (i.e., only a single bounce) for path tracing, meaning that the secondary diffuse rays do not spawn more rays. This means that we effectively get the same results as would be achieved in Figure 3.1 with maximum depth of 2, since the rasterization handles the first-hit and light contribution.

Sphere set proxies and voxel plane proxies were all generated using the algorithms described under Sections 4.2.2.1 and 4.2.3.1, respectively. Open-source reference implementations are available[2]. Simplified triangle meshes were generated using the open-source implementation[3] of the bounding mesh algorithm. Since the bounding mesh algorithm implementation does not respect texture coordinates, we create a new parameterization for the simplified mesh by projecting the original mesh onto it using Blender[4].

For each of the non-triangle proxy types, a custom hit-group was added to the renderer, consisting of an intersection shader and a closest-hit shader. The intersection algorithms were implemented as described in Sections 4.2.2.3 and 4.2.3.2. To keep the required memory throughput low, two strategies were employed: 16-bit floats were used in place of all uploaded floating point data (through the `VK_KHR_shader_float16_int8` extension), and as much memory access as possible was deferred to the closest-hit shader (instead of occurring in the intersection-shader).

To produce noise-free images for comparison purposes we render to 1024 SPP for all results, unless otherwise stated, by averaging 1024 frames of 1 SPP. To get consistent timing and test under dynamic scenarios, even though we do accumulate and don't actually have dynamic scenes, we rebuild the top-level acceleration structures every

[1] https://github.com/Shimmen/ArkoseRendererThesis
[2] https://github.com/Shimmen/ProxyGen
[3] https://github.com/gaschler/bounding-mesh
[4] https://www.blender.org/

frame. For all results below, the number of instances in the top-level acceleration structure is very low, so the rebuild time is insignificant (< 0.1 ms).

All timing tests are run on a Nvidia GeForce RTX 2080 Super at 1920×1080 pixels resolution, unless otherwise stated.

5.3 Diffuse GI performance in the RTX APIs

To understand the performance implications of rendering with diffuse rays compared to more coherent rays, an experiment was conducted whose results can be seen in Figure 5.2. To strictly evaluate the impact of ray coherency, great care was taken to eliminate other potential factors. To make sure the same number of rays hit the triangle mesh of interest (i.e., no miss-shader invocations), the mesh is closed and the camera is placed within the mesh. The mesh is also assigned a single uniform color value, so aspects such as texture sampling and texture memory fetching can be disregarded. Finally, each ray is traced from the G-buffer hit point and in the direction r_d, constructed as:

$$r_d = \frac{\mathbf{n} + \alpha^2 \mathbf{X}}{\|\mathbf{n} + \alpha^2 \mathbf{X}\|}, \tag{5.2}$$

where $\mathbf{n}$ is the smooth surface normal at the G-buffer hit point, and $\mathbf{X}$ is a uniformly random point on the unit sphere. α acts as a type of roughness parameter allowing us to vary the ray coherency for the purpose of testing. The reason α is squared is to give a more perceptibly linear roughness scale for the graphs where $0 < \alpha < 1$. The same remapping is performed in some common material models, such as the Disney BRDF [Bur12, p. 15].

The main reason (5.2) was chosen for this experiment is a combination of the fact that it trivially maps a single parameter (α) to ray coherency, and the fact that $\alpha = 1.0$ gives a distribution of rays equivalent to the photon distribution of a Lambertian diffuse surface [McG19c]. A consequence of this is that values of $\alpha < 1.0$ do not have meaningful interpretations. For example, $\alpha = 0$ (resulting in $r_d = \mathbf{n}$) is not similar to any physically plausible scattering profile. However, while it is not plausible it is similarly coherent to the scattering profile of a perfectly glossy/mirror surface. Substituting actual mirror scattering for the $\alpha = 0$ case does result in very similar performance.

Since the RTX-hardware uses BVHs internally, the rendering time should have a roughly logarithmic growth function, $f(n) = k \cdot \log(n)$, for the number of primitives n. As expected this can be observed in all graphs of Figure 5.2. However, very different values of k can also be observed, which result in quite significant differences in rendering time for large number of triangles. Since the number of rays that hit the mesh is held constant and the same constant color is used for all mesh hit points, the difference in speed must be attributed to BVH traversal and indexed vertex data lookup. Profiling further reveals that much of the active GPU time under

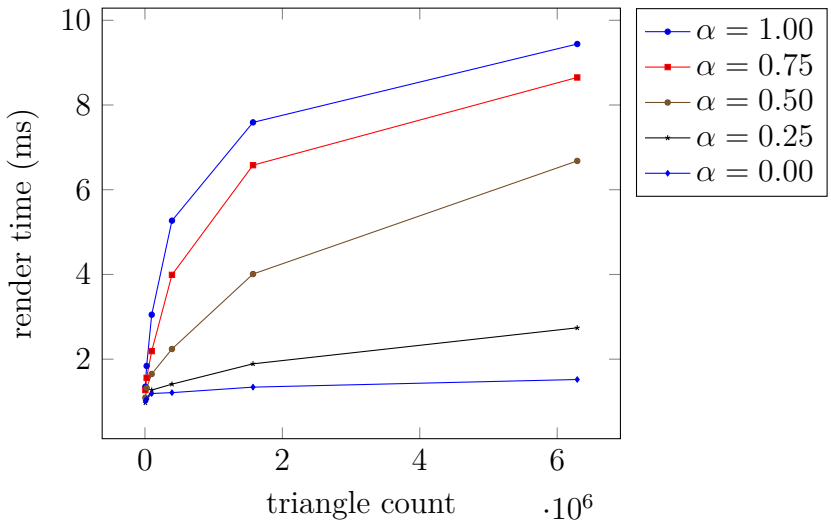

Figure 5.2: Rendering time of a single triangle mesh with increasing subdivision for varying values of α (5.2) at 1 SPP. It is clear that $O(\log\ n)$ behaviour can be observed in all cases due to the acceleration structures, but with different characteristics.

`vkCmdTraceRaysNV` is spent waiting for memory and that it is more significant for greater α.

These findings are in line with conclusions of previous work regarding memory access speed (see Section 2.2), and indicate that geometry representations with a smaller memory footprint may improve rendering performance for ray traced diffuse global illumination. From this experiment we know this holds for triangle-based geometry representations, but it is conceivable that a similar relation exist for other types of geometry. This tells us that it at least should be possible to achieve faster rendering times (errors aside) by decreasing the geometry memory footprint, as theorized when considering the choice of proxy types.

5.4 Evaluation of proxy geometry

In this section we will analyze each of the tested proxy geometry types and evaluate them according to the criteria set up in Section 1.2. The proxy geometries will also be compared to each other for the benefit of understanding their relative strengths and weaknesses.

The scene BunnyCorner[5] is used for direct comparisons between different proxy types and number of primitives. Using the original geometry 2.30 GRays/s can be achieved for diffuse global illumination and will be used as reference for all future comparisons under this section. Additionally, all error maps under this sub-section are relative to the indirect-light-only reference (Figure 5.3, bottom left), and use the same range so they can be directly compared to each other.

[5]Bunny model (Stanford Bunny) from *Computer Graphics Archive* [McG17].

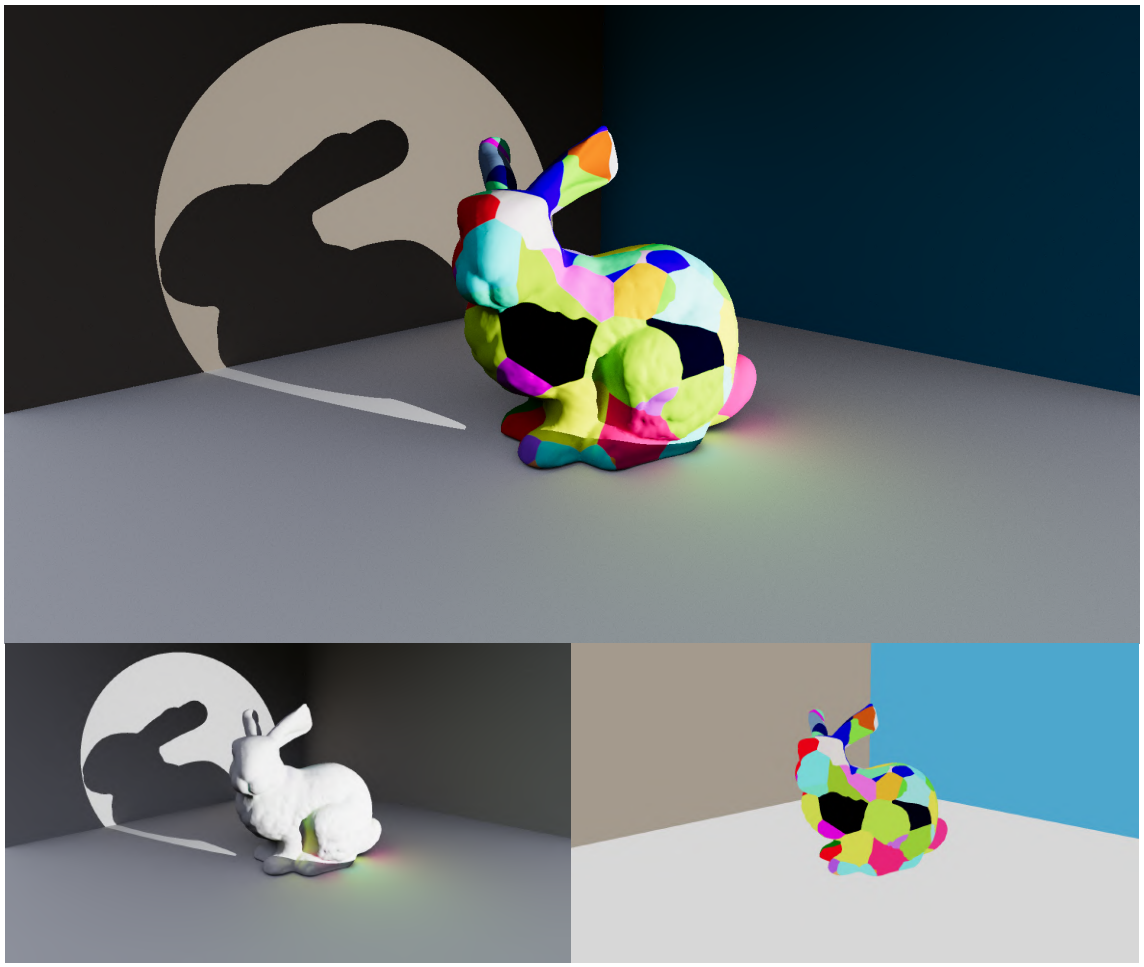

Figure 5.3: Reference render of BunnyCorner without the use of proxy geometry (144 046 triangles, excluding the geometry of the room, at 2.30 GRays/s). Top: with global illumination, bottom left: indirect light only, bottom right: visualization of the geometry.

5.4.1 Simplified triangle meshes

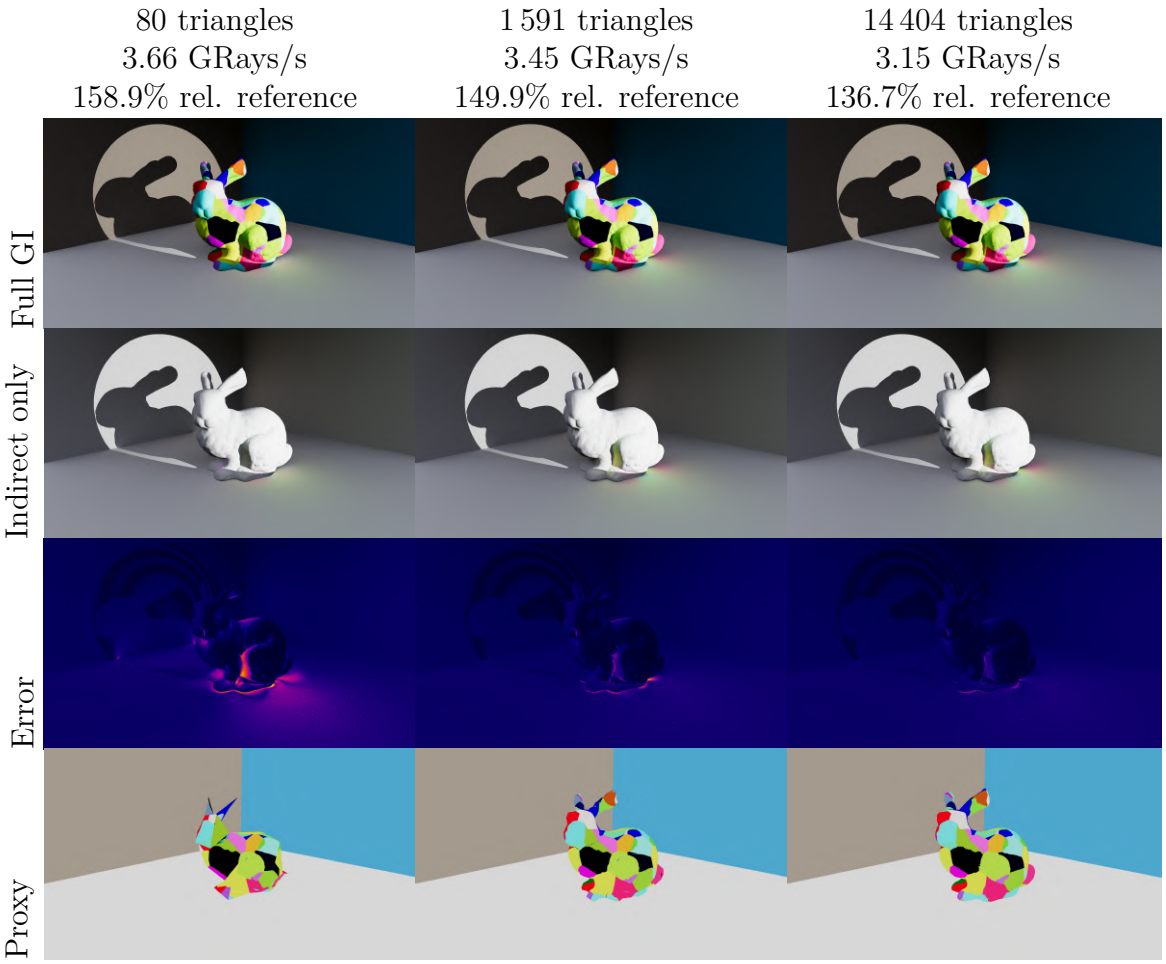

Figure 5.4: Visual quality and rendering time for simplified triangle mesh proxies, for a varying number of triangle primitives. The number of triangles does not include the geometry of the enclosing room.

Figure 5.4 shows rendered images and rendering times for the simplified triangle mesh proxies of the BunnyCorner scene. Disregarding the additional textures and potential miss-shader invocations, rendering times should be similar to the ones shown in Figure 5.2 for $\alpha = 1.0$. While the number of primitive triangles in the original reference geometry is quite low (144 046 triangles, excluding the room geometry), there is still a clear increase in rays per second, as number of simplified triangles decrease, as can be seen in Figure 5.4. More drastic increases would be expected on more complex scenes with more triangles, and this will be explored in Section 5.4.4.

The error maps in Figure 5.4 show a clear increase in error as the number of triangle primitives decrease. While there are some observable large-scale effects, the largest errors exist on the floor close to where the bunny model sits. Comparing to the reference it is clear that there is a lack of darkening in more confined spaces, i.e., ambient occlusion. As discussed in Section 4.1 this is to be expected, since having

all proxy geometry be bounded by the original geometry means ambient occlusion is underestimated.

5.4.2 Sphere set

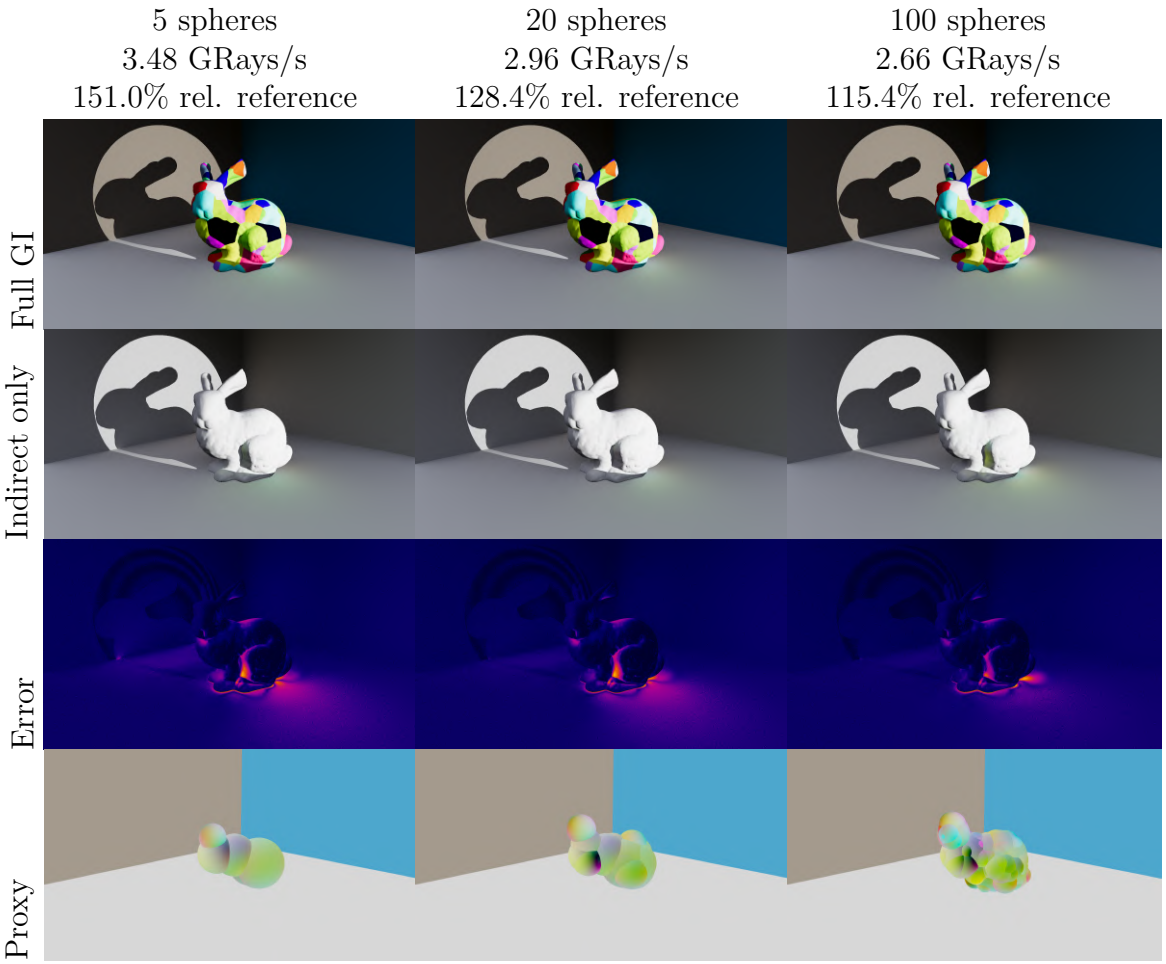

Figure 5.5: Visual quality and rendering time for simplified sphere set proxies, for a varying number of sphere primitives.

Figure 5.5 shows rendered images and rendering times for the sphere set proxies of the BunnyCorner scene. Using sphere set proxy geometry it is possible to achieve faster rendering times compared to the original geometry; however, there is a large amount of error in all maps which does not seem to decrease significantly with an increasing primitive count. Of course, as the number of primitives approaches infinity the error will tend to zero, but by extrapolating the observed data points it is clear that there is a reasonable range of number of primitives that can yield beneficial performance (i.e., faster than reference).

There are multiple possible reasons as to why the error does not decrease significantly within the evaluated range from 5 to 100 spheres:

1. the volumetric nature of spheres,

2. sub-optimal sphere set generation,

3. poor color encoding.

As discussed in Section 4.2.2 the volumetric nature of sphere can in theory be beneficial since a small amount of primitives can effectively fill a mesh. From the proxy visualization (bottom row Figure 5.5) it is clear that a small amount of spheres can roughly approximate the shape, however it is also clear that small features are often left out. For example, the ears are not represented in any of the three versions, and not until we have 100 spheres are the smaller features near the ground represented.

This also ties into the second point: sub-optimal sphere set generation. The algorithm used to generate sphere set representations (described in Section 4.2.2.1) is highly approximative and stochastic, and provides no guarantees that an optimal solution is found. The sphere teleportation step, specifically, is important for avoiding local minimas, but also introduces a lot of noise into the algorithm. It is very possible that a better algorithm could improve the errors.

Finally, the proxy visualization reveals that the ℓ-2 spherical harmonics color encoding fails to capture much of the relevant color information of the original geometry. The BunnyCorner scene has much high-frequency color, which is something that spherical harmonics is not great at capturing. Increasing the degree (ℓ) of the spherical harmonic encoding might improve this, but this will also increase the memory footprint which could be detrimental to the performance. In scenes with more low-frequency color the encoding can work better (see Section 5.4.4).

A single sphere occupies 8 bytes in GPU memory, or 80 bytes including SH color. This means that the geometry and colors of a 5-sphere set occupies no more than 400 bytes. Considering the clear correlation between memory footprint and performance as shown in Figure 5.2 it is curious that this geometry does not render faster. More so, 400 bytes should also be able to fit in all relevant GPU caches. Comparing to the simplified triangle meshes above, the performance is most similar that of the 1 591-triangle mesh, which occupies roughly 57KB for just the vertex position data (which is all that is required for intersection testing). Profiling reveals that much of the active GPU time under `vkCmdTraceRaysNV` is spent waiting for memory. This is similar to what was observed for triangle geometry as well, however, in this case it is an even larger fraction of the time.

5.4.3 Voxel planes

Figure 5.6 shows rendered images and rendering times for the voxel plane proxies of the BunnyCorner scene. As can be seen in the figure voxel plane proxy geometry is slower to render than the reference triangle geometry for all tested number of primitives.

Since voxel planes and spheres both require custom intersection shaders it can be interesting to compare them to each other to understand their relative performance characteristics. The main difference between them is the amount of data required to perform the intersection tests. A sphere is encoded as a 3D point and a radius,

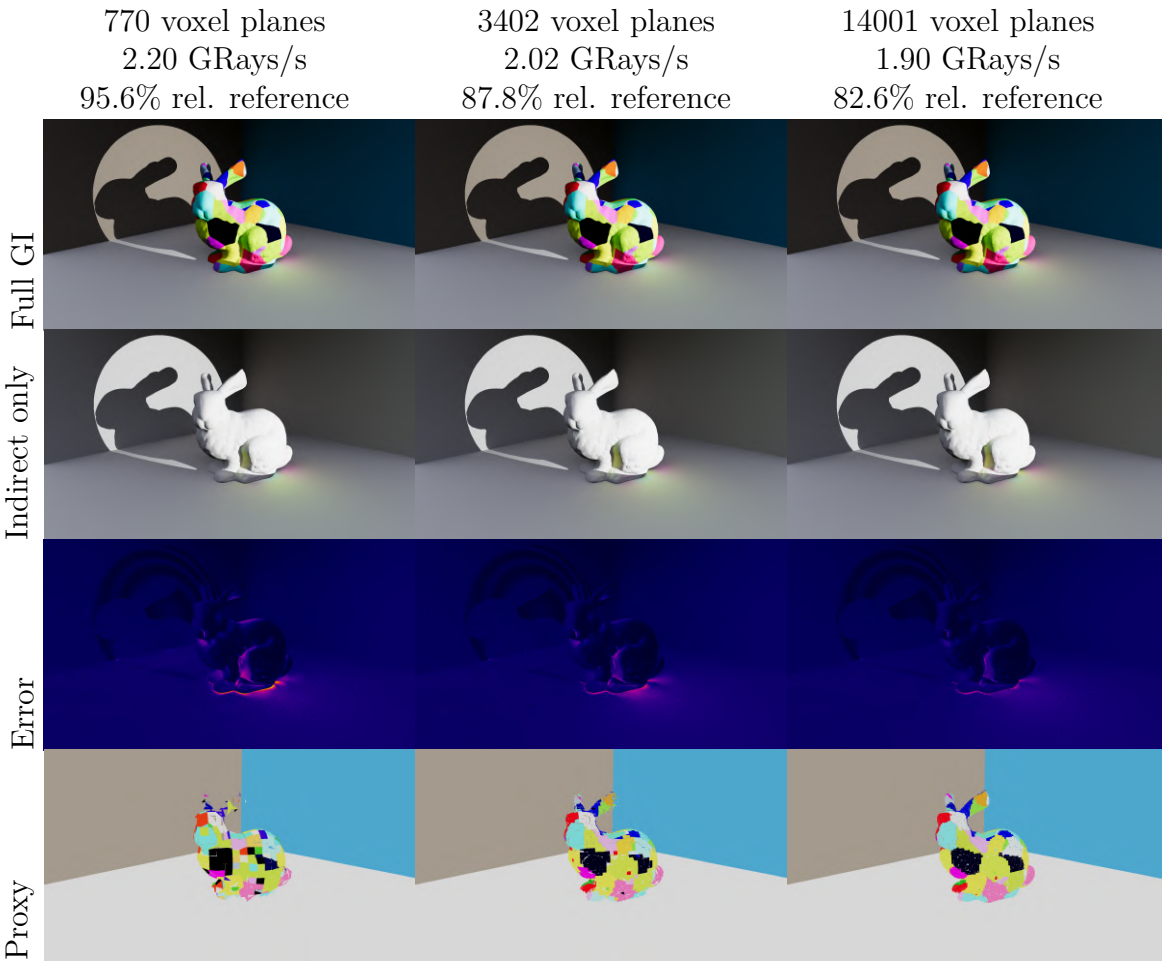

Figure 5.6: Visual quality and rendering time for simplified voxel plane proxies, for a varying number of voxel plane primitives.

while a voxel plane is encoded as a 3D surface normal, a distance from origin, a color index, and an AABB. Ignoring the AABB the representations require a similar number of bytes to encode. Since the invocation of an intersection shader occurs after a positive ray–AABB test some information about the AABB must be known at that point; however, no such information is passed to the intersection shader. If more information about the AABB was passed to the intersection shader it would be possible to render voxel planes without passing in a separate but identical set of AABBs. Assuming memory footprint is the most significant factor to rendering performance, similar times to those of sphere set proxies might be achievable.

Using voxel planes it is possible to get errors considerably smaller than what is achievable for sphere sets, for reasonable number of primitives. The rationale for what is reasonable is similar to what was argued for the sphere set proxies: using much less than 770 voxel planes results in very broken geometry and therefore also rendered images with little resemblance to the reference images. Since all tested number of voxel planes results in slower rendering we also see little reason to test higher numbers.

Within the evaluated range, as the number of voxel planes increase the proxy geometry approximates the original geometry increasingly well. However, as the number of voxel planes decrease the gaps between the individual planes increase. This is especially noticeable at areas of high curvature, such as the ears or feet of the bunny model (e.g., see Figure 5.6, bottom left).

5.4.4 More complex scenes

While the previous results in the BunnyCorner scene show how error and rendering times relate to proxy type and number of primitives, the scene does not accurately represent common rendering scenarios. For example, there is only a single object with proxy geometry, so no inter-proxy light interactions occur. For this purpose we created the scene SmallRoom[6]. Both the BunnyCorner and SmallRoom scenes also happen to be indoor scenes, with little light contribution from the environment. The RockyLandscape[7] scene was created to show a more realistic outdoors environment; additionally it has a significantly larger amount of input primitives than all the previously tested scenes.

Figure 5.7 shows the SmallRoom scene rendered using original geometry and all proxy types (the number of primitives for each proxy type was chosen arbitrarily, so direct comparison between the proxy techniques in this figure is not advised). Compared to BunnyCorner this scene has significantly more uniform colors, and looking at the sphere proxy it is evident that the SH color encoding performs better in this scene. However, the ambient occlusion is still problematic for the sphere set, which results in the sphere set having the most amount of error of the compared techniques for this scene. The voxel plane proxy achieves smaller errors than the sphere set proxy, but also at speeds significantly slower than the original geometry. Finally, the simplified triangle proxy renders with the smallest errors, and at significantly faster speeds compared to the original geometry: approximately 1.63 times the speed.

For the RockyLandscape scene the number of primitives were adjusted by eye so roughly similar errors could be achieved for the different proxy types. While not very precise, this allows us to draw clearer conclusions regarding rendering times for the different proxy types. Rendered images and rendering times can be seen in Figure 5.8. As before, both the simplified triangle proxies and sphere set proxies achieve faster rendering times than the reference. However, for this scene the voxel plane proxies are marginally faster. The reason for this could probably mostly be attributed to the significant number of input triangles, and relative to the other two proxy types, voxel planes are still less than half as fast. Also for the RockyLandscape scene most of the error can be attributed to an underestimated ambient occlusion.

[6]Barrel models from `https://3dmodelhaven.com/`. The room model is Cornell Box from *Computer Graphics Archive* [McG17].

[7]All models from Quixel `https://quixel.com/`.

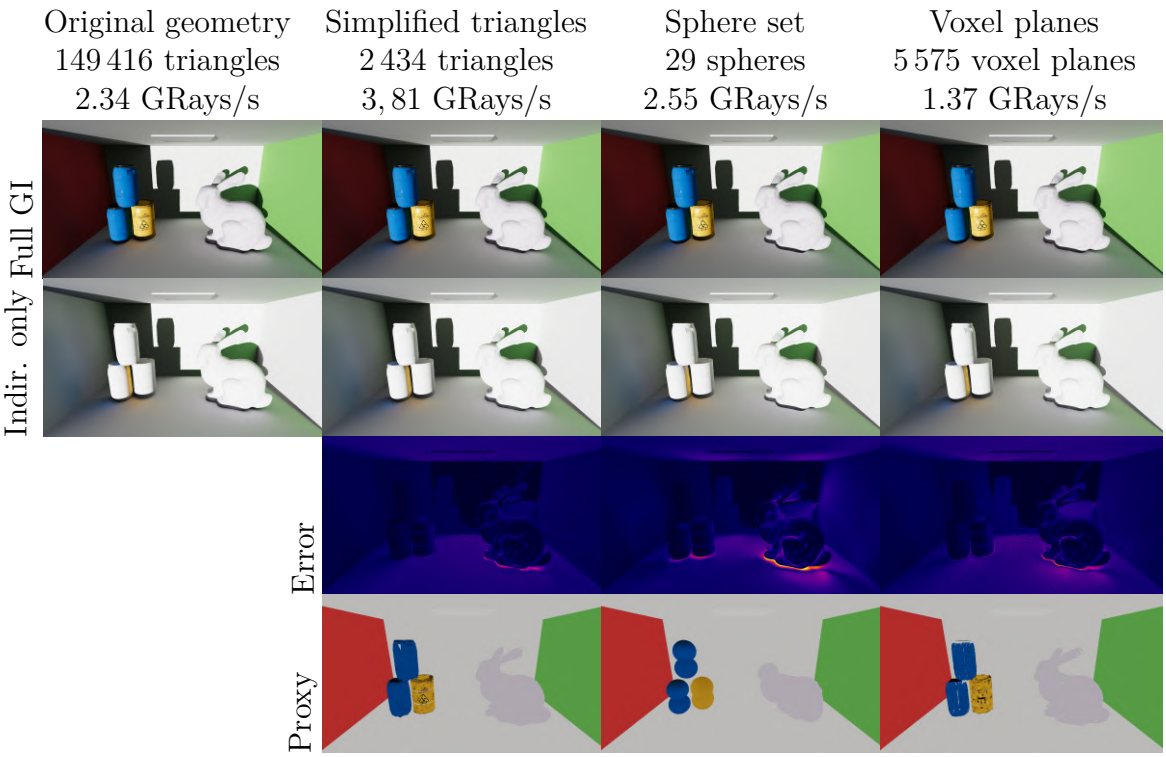

Figure 5.7: Sample of results rendering the SmallRoom scene for each of the proxy geometry types and reference.

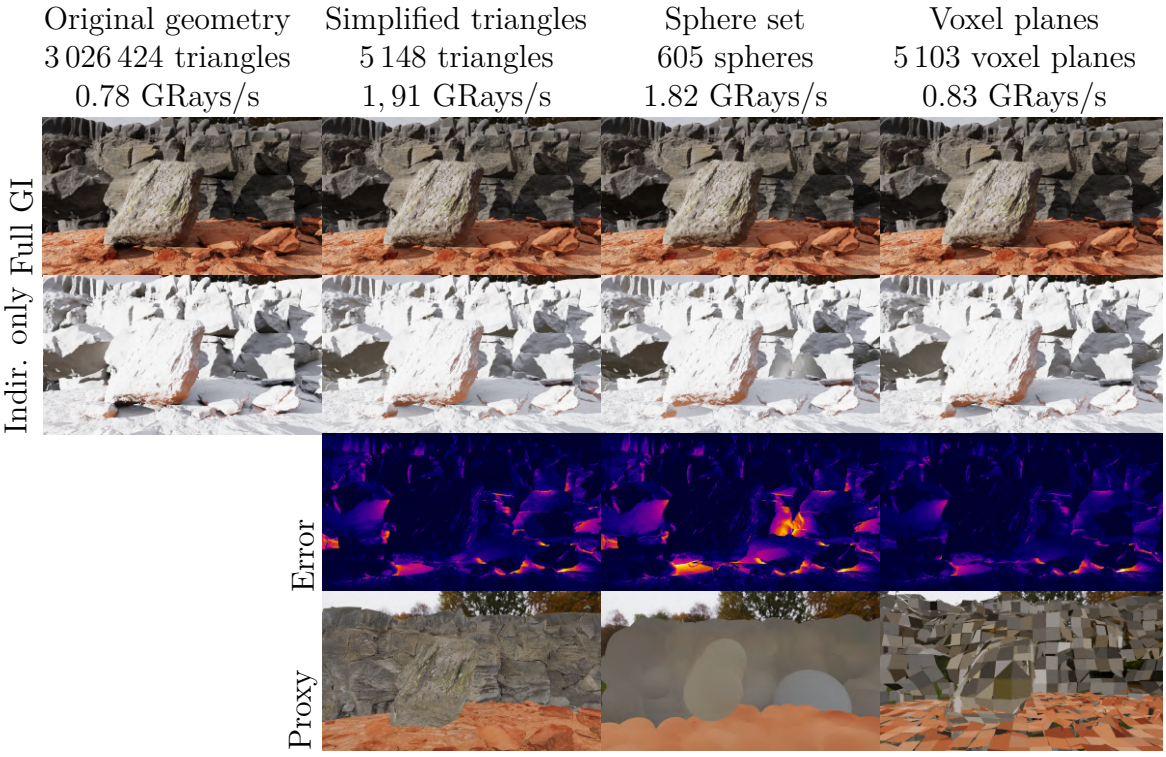

Figure 5.8: Sample of results rendering the RockyLandscape scene for each of the proxy geometry types and reference.

5.4.5 Compensating with ambient occlusion

Due to the bounding requirement (Item 3 Section 4.1), diffuse global illumination rendered using conforming proxy geometry will have underestimated ambient occlusion. This was initially theorized and later confirmed, as the effect is clearly visible in most results. However, approximations of ambient occlusion can be evaluated separately from diffuse global illumination and later combined to produce a potentially more accurate end result. In this section we will briefly explore the effects of adding separate ambient occlusion on top of the results already shown above. Since this type of ambient occlusion already exists in many real-time renderers we will not consider the additional rendering time imposed by it. The ambient occlusion for this experiment is implemented using ray tracing, by tracing rays against the original geometry, but ignoring objects which do not have a proxy representation (i.e., background walls and floor). Parameters of the ambient occlusion system are manually adjusted to fit the reference.

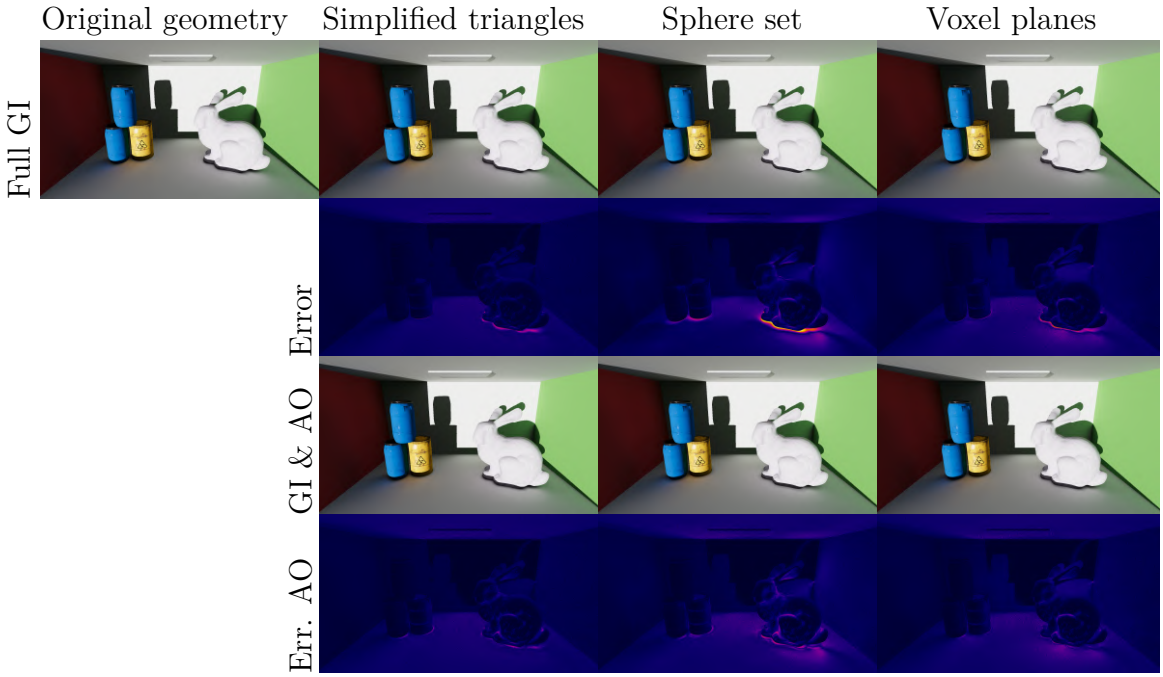

Figure 5.9: Renders of SmallRoom with and without added ambient occlusion (AO). Proxy geometry used is the same as in Figure 5.7.

Figure 5.9 shows the impact of rendering SmallRoom with and without added ambient occlusion (AO). For all of the proxy types the addition of ambient occlusion visibly improves the error. This is very clear from the error maps (row 2 and 4), but is also visible in the full global illumination with ambient occlusion (row 3) in comparison to the reference. While this is just an example scene, it clearly highlights what was previously just theorized: significant amounts of error introduced by using proxy geometry can be attributed to the underestimated ambient occlusion.

In Section 4.1 we argued that underestimating ambient occlusion, as a consequence of the bounding requirement, will result in artifacts, but that these artifacts are

less severe than what is to be expected if we do not ensure the requirement, as seen in Figure 4.1. One of the arguments was that the underestimation of ambient occlusion could in theory be complemented by a separate ambient occlusion system. This experiment shows that this works in practice, and we would therefore argue that the requirement is both important to ensure, and that it does not cause a significant problem.

6

Conclusion

6.1 Discussion

The results of this thesis show that using proxy geometry for rendering diffuse global illumination is both possible and can result in faster rendering times. Comparing the three different proxy geometry types evaluated for this thesis it is evident that simplified triangle meshes show the most potential. The triangle proxies are faster in all cases and render with the smallest errors for reasonable amounts of simplification. Additionally, for all tested scenes there is a simplified triangle mesh representation that has lower errors than all other proxy types and that is faster than the reference. This cannot be said for any of the other proxy types.

Regarding performance it is clear that triangle meshes have a unique advantage on current hardware and APIs. This is likely partially due to the custom hardware for ray–triangle intersection, but it is hard to evaluate the exact impact of the invocation compared to other aspects. As previously discussed, one of the most significant aspects for GPU ray tracing performance is memory latency and bandwidth. Beyond previous work, we can see this behaviour for triangles in Figure 5.2, and for non-triangle geometry by comparing the performance of sphere set and voxel plane proxies. However, by comparing the performance of 100 spheres in Figure 5.5 with 80 triangles in Figure 5.4 it is clear that the characteristics are vastly different for the two types, since the triangles are both faster and have a larger memory footprint. Furthermore, profiling reveals that stall times while waiting for memory is lower for triangle geometry despite it having a larger memory footprint. This could be an indication that some special purpose caching or other mechanism may occur specifically for triangle geometry, but more evidence is needed before something more conclusive can be said.

Beyond performance reasons it is also clear that triangles are more versatile for use as proxy geometry. While spheres have the advantage of being volumetric, it also gravely restricts the type of objects that can be represented as it with a small number of primitives. Objects of large volume to surface area ratio, such as the Stanford Bunny or a barrel, are quite trivially fitted by a few spheres, while objects with low ratio (i.e., thin objects), such as a curtain or a flat wall, need an infeasibly large amount of spheres to be represented. The inverse can be said about triangles: a thin wall can be represented by a few triangles, but a round object needs many

triangles. Similarly, the way voxel planes are defined means that only a single plane of a single color can exist within the bounds of a voxel in the grid. This enforces hard restrictions on the grid resolution, meaning for a thin wall to be captured by a grid of voxel planes, the grid resolution must be high enough so at least two voxels cover the thickness of the wall.

As shown, from a performance perspective non-triangle geometry has a disadvantage compared to fully hardware supported triangle geometry. With that said, there are still valid reasons to use other geometry primitives than triangles, and it is fully supported by the APIs. While fully supported, there exist API constraints that potentially limit the performance: voxel planes could, in theory, have similar performance characteristics to that of sphere set but is hindered by the lack of AABB information in the intersection shaders. Exposing more information to intersection shaders, such as the AABB definition, could greatly improve voxel plane performance. A less intrusive API change could be to adjust the min and max distances for the current ray to be AABB relative instead of being globally defined. These values are already exposed in the API as `gl_RayTminNV`, which represents the t-min passed to the `traceNV` invocation, and `gl_RayTmaxNV`, which represents the distance to the closest primitive intersected so far. From the perspective of an intersection shader the values could be set to the current AABB hit min and max values, which would reveal more information about the AABB without adding more data to the intersection shader API.

6.2 Conclusion

We have explored three different types of proxy geometry for use in rendering ray traced diffuse global illumination. Overall we have shown that using proxy geometry this way is possible, and that higher rendering speeds can be achieved. As accuracy is sacrificed for greater rendering speed, the visual quality of rendered images will decrease when using proxy geometry. Since a judgment has to be made in the trade-off between speed and accuracy—two separate metrics—an objective evaluation is not possible. With that said, we believe simplified triangle meshes show the most potential for this purpose.

For all tested scenes there is a simplified triangle mesh representation that has visibly lower errors than all other proxy types and that is faster than the reference. This cannot be said for any of the other proxy types, and is the reason we consider this proxy type favourable. Sphere set proxies could always render faster than the original geometry, but introduced significant errors. With voxel plane proxies lower errors could in general be achieved, compared to the sphere set proxies, however, lower rendering speeds was achieved in most evaluated scenes, relative to the input geometry. Because of this, they should not be considered usable for the purpose of this thesis, but future hardware and APIs may improve their performance.

Additionally we have shown that having proxy geometry be fully bounded by the original geometry introduces an underestimation of ambient occlusion, and that this error can be mitigated by complementing the proxy ray traced global illumination

with some type of separate ambient occlusion system. Proxy geometry that is not bounded, however, introduces irreversible artifacts.

Finally, we note that many more potential proxy geometry types exist, beyond the three evaluated for this thesis. While nothing specific can be said about untested proxy types, we believe triangle-based proxies are most promising, due to the evident advantage in rendering performance.

6.2.1 Future work

As previously mentioned this thesis is written at the advent of non-vendor-specific APIs for real-time ray tracing, but Nvidia is still the only vendor which sells GPU with ray tracing hardware to consumers. When more vendors provide GPUs with such hardware and capabilities it would be very interesting to analyse the potential differences in performance characteristics between the vendors' GPUs.

Simplified triangle meshes show much potential for rendering ray traced diffuse global illumination, as demonstrated in this thesis. However, there are some obstacles when it comes to generating bounded meshes. The bounding mesh algorithm is one of very few existing algorithms that can generate such meshes, but it has both missing features (texture coordinate awareness) and uses *only* edge contractions, which today is generally considered to be non-optimal. We think more research regarding generating bounded meshes is needed. Specifically we believe that incorporating bounding constraints into state-of-the-art texture aware mesh simplification algorithms (e.g., [Liu+17]) is a good avenue for research.

Bibliography

[ABI88] David Avis, Binay K. Bhattacharya, and Hiroshi Imai. "Computing the volume of the union of spheres". In: *The Visual Computer* 3 (1988), pp. 323–328. ISSN: 1432-2315. URL: https://doi.org/10.1007/BF01901190 (cit. on p. 19).

[Ake+18a] Tomas Akenine-Möller, Eric Haines, Naty Hoffman, et al. *Real-Time Rendering, Fourth Edition*. A K Peters/CRC Press, 2018. ISBN: 1138627003 (cit. on pp. 2, 7, 9, 27).

[Ake+18b] Tomas Akenine-Möller, Eric Haines, Naty Hoffman, et al. "Real-Time Rendering, Fourth Edition". In: 4th ed. Nov. 2018. Chap. Online chapter: Real-Time Ray Tracing version 1.4. URL: https://www.realtimerendering.com/Real-Time_Rendering_4th-Real-Time_Ray_Tracing.pdf (cit. on pp. 3, 28).

[AL09] Timo Aila and Samuli Laine. "Understanding the Efficiency of Ray Traversal on GPUs". In: *Proceedings of the Conference on High Performance Graphics 2009*. HPG '09. New Orleans, Louisiana: Association for Computing Machinery, 2009, pp. 145–149. ISBN: 9781605586038. DOI: 10.1145/1572769.1572792. URL: https://doi.org/10.1145/1572769.1572792 (cit. on pp. 5, 6).

[App68] 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: ACM, 1968, pp. 37–45. DOI: 10.1145/1468075.1468082. URL: http://doi.acm.org/10.1145/1468075.1468082 (cit. on p. 5).

[Bar+19] Colin Barré-Brisebois, Henrik Halén, Graham Wihlidal, et al. "Hybrid Rendering for Real-Time Ray Tracing". In: *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*. Ed. by Eric Haines and Tomas Akenine-Möller. Berkeley, CA: Apress, 2019, pp. 437–473. ISBN: 978-1-4842-4427-2. DOI: 10.1007/978-1-4842-4427-2_25. URL: https://doi.org/10.1007/978-1-4842-4427-2_25 (cit. on p. 28).

[BO02] Gareth Bradshaw and Carol O'Sullivan. "Sphere-Tree Construction Using Dynamic Medial Axis Approximation". In: *Proceedings of the 2002 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. SCA '02. San Antonio, Texas: Association for Computing Machinery,

2002, pp. 33–40. ISBN: 1581135734. DOI: 10.1145/545261.545267. URL: https://doi.org/10.1145/545261.545267 (cit. on pp. 7, 8, 19).

[Bur12] Brent Burley. *Physically-Based Shading at Disney*. Tech. rep. Walt Disney Animation Studios, 2012. URL: https://disney-animation.s3.amazonaws.com/library/s2012_pbs_disney_brdf_notes_v2.pdf (cit. on p. 29).

[Cra+11] Cyril Crassin, Fabrice Neyret, Miguel Sainz, et al. "Interactive Indirect Illumination Using Voxel Cone Tracing: A Preview". In: *Symposium on Interactive 3D Graphics and Games*. I3D '11. San Francisco, California: Association for Computing Machinery, 2011, p. 207. ISBN: 9781450305655. DOI: 10.1145/1944745.1944787. URL: https://doi.org/10.1145/1944745.1944787 (cit. on p. 7).

[CS15] Levan Chkhartishvili and Gokul Suryamurthy. "Volume of intersection of six spheres: A special case of practical interest". In: *Nano Studies* 11 (Jan. 2015), pp. 111–126 (cit. on pp. 19, 20).

[Den+17] Yangdong Deng, Yufei Ni, Zonghui Li, et al. "Toward Real-Time Ray Tracing: A Survey on Hardware Acceleration and Microarchitecture Techniques". In: *ACM Comput. Surv.* 50.4 (Aug. 2017). ISSN: 0360-0300. DOI: 10.1145/3104067. URL: https://doi.org/10.1145/3104067 (cit. on p. 6).

[DFM13] Michael J. Doyle, Colin Fowler, and Michael Manzke. "A Hardware Unit for Fast SAH-Optimised BVH Construction". In: *ACM Trans. Graph.* 32.4 (July 2013). ISSN: 0730-0301. DOI: 10.1145/2461912.2462025. URL: https://doi.org/10.1145/2461912.2462025 (cit. on p. 6).

[GFK15] Andre Gaschler, Quirin Fischer, and Alois Knoll. *The Bounding Mesh Algorithm*. Tech. rep. TUM-I1522. Technische Universität München, Germany, 2015. URL: https://mediatum.ub.tum.de/node?id=1255722 (cit. on pp. 7, 17, 18).

[GM05] Enrico Gobbetti and Fabio Marton. "Far Voxels: A Multiresolution Framework for Interactive Rendering of Huge Complex 3D Models on Commodity Graphics Platforms". In: *ACM Trans. Graph.* 24.3 (July 2005), pp. 878–885. ISSN: 0730-0301. DOI: 10.1145/1073204.1073277. URL: https://doi.org/10.1145/1073204.1073277 (cit. on p. 7).

[Gre03] Robin Green. *Spherical Harmonic Lighting: The Gritty Details*. Tech. rep. 2003. URL: https://basesandframes.wordpress.com/2016/05/11/spherical-harmonic-lighting-the-gritty-details/ (cit. on p. 21).

[HSK89] J. C. Hart, D. J. Sandin, and L. H. Kauffman. "Ray Tracing Deterministic 3-D Fractals". In: *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '89. New York, NY, USA: Association for Computing Machinery, 1989, pp. 289–296. ISBN: 0897913124. DOI: 10.1145/74333.74363. URL: https://doi.org/10.1145/74333.74363 (cit. on p. 8).

[III04] Jerry O. Talton III. *A Short Survey of Mesh Simplification Algorithms*. 2004. URL: http://jerrytalton.net/research/t-ssmsa-04/paper.pdf (cit. on pp. 7, 17).

[Kaj86] 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. URL: http://doi.acm.org/10.1145/15886.15902 (cit. on pp. 5, 10).

[Khra] Khronos Group. *GLSL_NV_ray_tracing*. URL: https://github.com/KhronosGroup/GLSL/blob/master/extensions/nv/GLSL_NV_ray_tracing.txt (cit. on p. 12).

[Khrb] Khronos Group. *VK_NV_ray_tracing*. URL: https://www.khronos.org/registry/vulkan/specs/1.2-extensions/man/html/VK_NV_ray_tracing.html (cit. on p. 12).

[LCP17] P. Lousada, V. Costa, and J.M. Pereira. "Bandwidth and memory efficiency in real-time ray tracing". In: *Journal of WSCG* 25 (Jan. 2017), pp. 49–58. URL: https://otik.uk.zcu.cz/bitstream/11025/26279/1/Lousada.pdf (cit. on p. 6).

[Liu+17] Songrun Liu, Zachary Ferguson, Alec Jacobson, et al. "Seamless: Seam erasure and seam-aware decoupling of shape from mesh resolution". In: *ACM Transactions on Graphics (TOG)* 36.6 (Nov. 2017), 216:1–216:15. ISSN: 0730-0301. DOI: 10.1145/3130800.3130897 (cit. on p. 43).

[LK10a] Samuli Laine and Tero Karras. "Efficient Sparse Voxel Octrees". In: *Proceedings of ACM SIGGRAPH 2010 Symposium on Interactive 3D Graphics and Games*. 2010 (cit. on p. 7).

[LK10b] Samuli Laine and Tero Karras. *Efficient Sparse Voxel Octrees – Analysis, Extensions, and Implementation*. NVIDIA Technical Report NVR-2010-001. Feb. 2010 (cit. on pp. 7, 22).

[Maj+19] Zander Majercik, Jean-Philippe Guertin, Derek Nowrouzezahrai, et al. "Dynamic Diffuse Global Illumination with Ray-Traced Irradiance Fields". In: *Journal of Computer Graphics Techniques (JCGT)* 8.2 (June 2019), pp. 1–30. ISSN: 2331-7418. URL: http://jcgt.org/published/0008/02/01/ (cit. on p. 3).

[McG17] Morgan McGuire. *Computer Graphics Archive*. https://casual-effects.com/data. July 2017. URL: https://casual-effects.com/data (cit. on pp. 30, 36).

[McG19a] Morgan McGuire. "The Graphics Codex". In: Edition 2.17. 2019. Chap. The Rendering Equation. URL: http://graphicscodex.com/index.php (cit. on p. 9).

[McG19b] Morgan McGuire. *The Graphics Codex*. Edition 2.17. 2019. URL: http://graphicscodex.com/index.php (cit. on p. 9).

[McG19c] Morgan McGuire. "The Graphics Codex". In: Edition 2.17. 2019. Chap. Materials. URL: http://graphicscodex.com/index.php (cit. on pp. 10, 29).

[Nah+11] Jae-Ho Nah, Jeong-Soo Park, Chanmin Park, et al. "T&I Engine: Traversal and Intersection Engine for Hardware Accelerated Ray Tracing". In: *ACM Trans. Graph.* 30.6 (Dec. 2011), pp. 1–10. ISSN: 0730-0301. DOI: 10.1145/2070781.2024194. URL: https://doi.org/10.1145/2070781.2024194 (cit. on p. 6).

[Nah+14] Jae-Ho Nah, Hyuck-Joo Kwon, Dong-Seok Kim, et al. "RayCore: A Ray-Tracing Hardware Architecture for Mobile Devices". In: *ACM Trans. Graph.* 33.5 (Sept. 2014). ISSN: 0730-0301. DOI: 10.1145/2629634. URL: https://doi.org/10.1145/2629634 (cit. on p. 6).

[NVI18] NVIDIA. *NVIDIA Turing GPU Architecture*. Accessed 2020-03-16. 2018. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf (cit. on pp. 3, 6, 12).

[PJH18] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. 3rd ed. Morgan Kaufmann, 2018. ISBN: 9780128006450. URL: http://www.pbr-book.org/ (cit. on pp. 5, 9–11).

[Pow09] M. J. D. Powell. *The BOBYQA algorithm for bound constrained optimization without derivatives*. Technical report, Department of Applied Mathematics and Theoretical Physics, University of Cambridge. 2009. URL: http://www.damtp.cam.ac.uk/user/na/NA_papers/NA2009_06.pdf (cit. on p. 20).

[PT02] Ioannis Pantazopoulos and Spyros Tzafestas. "Occlusion Culling Algorithms: A Comprehensive Survey". In: *Journal of Intelligent and Robotic Systems* 35 (Oct. 2002), pp. 123–156. DOI: 10.1023/A:1021175220384 (cit. on pp. 3, 7).

[Qui13] Inigo Quilez. *SH - visualizer*. Quilez' homepage: iquilezles.org/www/index.htm. 2013. URL: https://www.shadertoy.com/view/lsfXWH (cit. on p. 22).

[Ren+06] Zhong Ren, Rui Wang, John Snyder, et al. "Real-time Soft Shadows in Dynamic Scenes Using Spherical Harmonic Exponentiation". In: *ACM SIGGRAPH 2006 Papers*. SIGGRAPH '06. Boston, Massachusetts: ACM, 2006, pp. 977–986. ISBN: 1-59593-364-6. DOI: 10.1145/1179352.1141982. URL: http://doi.acm.org/10.1145/1179352.1141982 (cit. on pp. 8, 19).

[San+19] V.V. Sanzharov, A.I. Gorbonosov, V.A. Frolov, et al. "Examination of the Nvidia RTX". In: *Proceedings of the 29th International Conference on Computer Graphics and Vision*. Vol. 2485. Bryansk, Russia: CEUR Workshop Proceedings, 2019, pp. 7–12. URL: http://ceur-ws.org/Vol-2485/paper3.pdf (cit. on p. 6).

[Sch+17] Christoph Schied, Anton Kaplanyan, Chris Wyman, et al. "Spatiotemporal Variance-guided Filtering: Real-time Reconstruction for Path-traced Global Illumination". In: *Proceedings of High Performance Graphics*. HPG '17. Los Angeles, California: ACM, 2017, 2:1–2:12. ISBN: 978-1-4503-5101-0. DOI: 10.1145/3105762.3105770. URL: http://doi.acm.org/10.1145/3105762.3105770 (cit. on p. 3).

[Sil+14] Ari Silvennoinen, Hannu Saransaari, Samuli Laine, et al. "Occluder Simplification Using Planar Sections". In: *Comput. Graph. Forum* 33.1 (Feb. 2014), pp. 235–245. ISSN: 0167-7055. DOI: 10.1111/cgf.12271. URL: https://doi.org/10.1111/cgf.12271 (cit. on pp. 3, 7, 17).

[Spj+09] Josef Spjut, Andrew Kensler, Daniel Kopta, et al. "TRaX: A Multicore Hardware Architecture for Real-Time Ray Tracing". In: *Trans. Comp.-*

Aided Des. Integ. Cir. Sys. 28.12 (Dec. 2009), pp. 1802–1815. ISSN: 0278-0070. DOI: 10.1109/TCAD.2009.2028981. URL: https://doi.org/10.1109/TCAD.2009.2028981 (cit. on p. 6).

[Vii+18] Timo Viitanen, Matias Koskela, Pekka Jääskeläinen, et al. "PLOCTree: A Fast, High-Quality Hardware BVH Builder". In: *Proc. ACM Comput. Graph. Interact. Tech.* 1.2 (Aug. 2018). DOI: 10.1145/3233309. URL: https://doi.org/10.1145/3233309 (cit. on p. 6).

[Wan+06] Rui Wang, Kun Zhou, John Snyder, et al. "Variational Sphere set Approximation for Solid Objects". In: *The Visual Computer* (Aug. 2006). URL: https://www.microsoft.com/en-us/research/publication/variational-sphere-set-approximation-solid-objects/ (cit. on pp. 7, 8, 19–21).

[Whi80] Turner Whitted. "An Improved Illumination Model for Shaded Display". In: *Commun. ACM* 23.6 (June 1980), pp. 343–349. ISSN: 0001-0782. DOI: 10.1145/358876.358882. URL: http://doi.acm.org/10.1145/358876.358882 (cit. on p. 5).

[Woo04] Sven Woop. *A Ray Tracing Hardware Architecture for Dynamic Scenes*. Tech. rep. Saarland University, 2004 (cit. on p. 6).

[YKL17] Henri Ylitie, Tero Karras, and Samuli Laine. "Efficient Incoherent Ray Traversal on GPUs through Compressed Wide BVHs". In: *Proceedings of High Performance Graphics*. HPG '17. Los Angeles, California: Association for Computing Machinery, 2017. ISBN: 9781450351010. DOI: 10.1145/3105762.3105773. URL: https://doi.org/10.1145/3105762.3105773 (cit. on p. 6).