Tampere University

Petrus Kivi

RAY TRACING METHODS FOR POINT CLOUD RENDERING

Faculty of Engineering and Natural Sciences
Masters thesis
November 2019

ABSTRACT

Petrus Kivi: Ray tracing methods for point cloud rendering
Masters thesis
Tampere University
Degree Programme in Science and Engineering
November 2019

State of the art scanning and capturing devices are able to produce surface point cloud models of a wide range of real world objects. The visualization and rendering of enormous point clouds with millions or billions of points is demanding. VR- and AR-applications can utilize embedded real world objects in generating visually pleasing and immersive virtual worlds. In order to achieve convincing real life equivalents in VR, rendering techniques that can replicate realistic material and lighting effects are needed. This can be achieved by utilizing ray tracing methods to render the virtual world onto a monitor or a head-mounted display.

Virtual reality applications need real-time stereoscopic rendering with high frame rates and resolution to produce a realistic and comfortable experience. This sets high demands on a point cloud ray tracing pipeline, which needs efficient intersection testing between rays and point cloud models. An easily intersectable global surface can be reconstructed from the point cloud model with, e.g., triangle mesh reconstruction. However, this can be computationally demanding and even wasteful if parts of the model are out of view or occluded. Direct point cloud ray tracing methods consider local features of the point cloud to generate intersectable surfaces only when needed.

In this thesis, we survey and compare different methods for directly ray tracing point cloud models without global surface reconstruction. Methods are compared with asymptotic complexity analysis and it is concluded that direct ray tracing of point clouds can be computationally more efficient compared to global surface reconstruction.

Keywords: point cloud, ray tracing, reconstruction, acceleration structure, rendering equation

TIIVISTELMÄ

Petrus Kivi: Säteenjäljitysmenetelmiä pistepilvien renderointiin
Diplomityö
Tampereen yliopisto
Teknis-luonnontieteellinen koulutusohjelma
Marraskuu 2019

Nykyaikaiset 3D-ympäristön kuvantamislaitteet pystyvät tuottamaan virtuaalisia pistepilvimalleja monenlaisista tosimaailman kappaleista. Miljoonia tai miljardeja datapisteitä sisältävien valtavien pistepilvimallien visualisointi ja renderointi on laskennallisesti työlästä. Virtuaalitodellisuussovellukset, kuten VR- ja AR-ohjelmat, voivat upottaa todellisten kappaleiden malleja virtuaalitodellisuuteen, jolla saadaan aikaiseksi visuaalisesti miellyttäviä ja immersiivisiä virtuaalikokemuksia. Luodakseen uskottavan kopion oikeasta kappaleesta VR-todellisuuteen, renderointitekniikan täytyy tukea realistisia materiaalin ja valon välisiä vuorovaikutuksia. Tämä pystytään saavuttamaan hyödyntämällä säteenjäljitystä, joka renderoi virtuaalimaailman tietokoneen tai VR-kypärän näytölle.

VR-sovellukset vaativat reaaliaikaista, korkean virkistystaajuuden stereoskooppista renderointia luodakseen todentuntuisen ja miellyttävän kokemuksen. Tämä asettaa tiukat vaatimukset säteenjäljitysrenderoijalle, mikä tarkoittaa että säteiden ja pistepilven välisten leikkauskohtien etsiminen täytyy olla tehokasta. Helposti säteenjäljitettävä pinnan globaali rekonstruktio voidaan luoda esimerkiksi kolmiomallirekonstruktion avulla. Globaali rekonstruktio voi kuitenkin olla laskennallisesti vaativaa ja epätaloudellista, jos osa mallista on katsekulman ulkopuolella tai peitossa. Menetelmät, jotka suoraan säteenjäljittävät pistepilvimalleja ilman globaalia pinnanrekonstruktiota, luovat paikallisen säteenjäljitettävän pinnan vain tarvittaessa.

Tässä opinnäytetyössä kartoitetaan ja vertaillaan eri menetelmiä pistepilvimallien suoraan säteenjäljitykseen ilman globaalia pinnan rekonstruktiota. Eri menetelmiä vertaillaan laskennallisen asymptoottisen kompleksisuuden avulla. Yhteenvetona todetaan, että pistepilvien suora säteenjäljitys voi olla laskennallisesti tehokkaampaa kuin globaalin pinnan rekonstruointi.

Avainsanat: pistepilvi, säteenjäljitys, rekonstruointi, kiihdytysrakenne, renderointiyhtälö

PREFACE

This thesis was done while working as a research assistant in the CPC-VGA group at the Computing Sciences unit of Tampere University. The making of the thesis was supported by the EC-SEL JU project FitOptiVis (project number 783162).

Thank you to both of my supervisors Dr. Markku Åkerblom and Dr. Markku Mäkitalo for guiding me diligently during this thesis. They gave me insight and motivation to continue with the thesis subject through all its trials and tribulations.

I would also like to thank Prof. Pekka Jääskeläinen for giving the opportunity to work on this thesis subject. I hope that the trust he has put in me will come to fruition through this thesis and the contributions following it.

A big thanks goes to all my colleagues in the CPC-VGA group at Tampere University, especially Matias Koskela for introducing me to the wondrous world of computer graphics. To my family, friends, and relatives, I want to express my deepest gratitude for making me the person capable of facing and conquering even the scariest obstacles in life. Mom, Dad, and Sohvi, thank you!

Riina, without you I wouldn't have made it this far. You held the fort when I couldn't, as always. Thank you, my love!

Tampereella, 4th November 2019

Petrus Kivi

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

$\cdot$	Dot product in $\mathbb{R}^n$
$\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{u}, \boldsymbol{v}$	Vector-valued variables
B	Number of recursive ray bounces
$\mathbb{C}$	Complex numbers
$\mathbb{C}^{m \times n}$	Complex-valued $m \times n$ dimensional matrices
C^{-1}	The set of piece-wise continuous functions
C^0	The set of continuous functions
C^n	The set of n-times differentiable functions
O	Big O asymptotic complexity
δ_{ij}	Kronecker delta
$\mathcal{E}^n$	Orthonormal basis of $\mathbb{R}^n$
k	Number of points in a regularly sampled interval
I	Number of iterative steps
K	Number of objects within a neighborhood
M	Number of pixels on a virtual screen
$\mathbb{N}$	Natural numbers
N	Number of points in a point cloud
$\mathcal{P}(X)$	Probability of outcome X
$\mathbb{R}$	Real numbers
$\mathbb{R}^n$	n-dimensional Euclidian space
$\mathbb{R}^{m \times n}$	Real-valued $m \times n$ dimensional matrices
T_i^j	Execution time of line i in Algorithm j
θ	Weighting function $\mathbb{R} \to \mathbb{R}$
$\mathbb{V}$	An arbitrary vector space
$\mathbb{Z}_+$	Non-negative integers

2D	Two-dimensional
3D	Three-dimensional
AO	Ambient occlusion

BRDF	Bidirectional reflectance distribution function
BSDF	Bidirectional scattering distribution function
BTDF	Bidirectional transmittance distribution function
BVH	Bounding volume hierarchy
CSM	Caustic sample map
EWA	Elliptical weighted average
fps	Frames per second
GI	Global illumination
GPU	Graphics processing unit
k-NN	k nearest neighbors
LIDAR	Light detection and ranging
LOD	Level of detail
LTE	Light transport equation
MLS	Moving least squares
NEE	Next event estimation
PBR	Physically based rendering
PDF	Probability density function
RGB	Red-green-blue
RGB-D	Red-green-blue-depth
SDF	Signed distance function
spp	Samples per pixel

1 INTRODUCTION

Rendering in computer graphics is the process of producing two-dimensional (2D) images of a virtual scene onto a monitor. The analogous process in the real world is taking a photograph: a camera has a small hole and a lens that capture the light bouncing off or passing through three-dimensional (3D) objects onto a sensor or film. The different parts of the camera have a certain location and angle with respect to each other and the world, which determine how the 3D world is projected onto the film or sensor. In computer graphics rendering, the *virtual camera*, especially its center, is the abstraction used for the camera hole or lens through which the light passes. The *virtual screen* is the counterpart to the sensor or film of a real camera. The real world camera has the sensor setup behind the camera hole or lens where as in rendering, the virtual screen is in front of the virtual camera, which is depicted in Figure 1.1 [49, pp. 5–6].

The *rendering pipeline* describes the different parts of the process of generating a 2D image of a 3D virtual scene within a single *image frame* of a continuous stream of rendered images. A generic rendering pipeline consists of three stages: the application, geometry and pixel stages. The application stage sets up the objects and lights within a virtual scene for the current frame. The geometry stage transforms objects, lights and the virtual camera into the same coordinate system, then applies the scene lighting to the geometry of the objects, and projects the objects onto the virtual screen. Lastly, the pixel stage resolves which projected objects are visible in each pixel and applies a *texture color*, i.e., the main color of an objects to the pixels [4, pp. 11–27].

In this thesis, we mostly focus on the geometry stage, especially the process of lighting objects and projecting them onto the virtual screen with *perspective projection*. Figure 1.2 shows the overview of the geometry stage, which encompasses the two main categories for projecting 3D geometry onto a 2D virtual screen: *rasterizing* methods and *ray tracing* methods. Rasterizing methods iterate through all possibly visible geometry and project them onto the virtual screen with perspective projection. Lighting is produced on the possibly visible objects by using approximate shading techniques like *Phong shading* [4, pp. 20–22].

Ray tracing methods go through all the virtual screen pixels casting a ray from the virtual camera through the pixels and resolving either the closest or multiple ray-object intersections in the scene. Lighting can be calculated in a similar fashion to rasterizing with approximate shading, or by casting more rays from the closest intersection towards the scene lights and other geometry. Rays casted from the virtual camera center through the

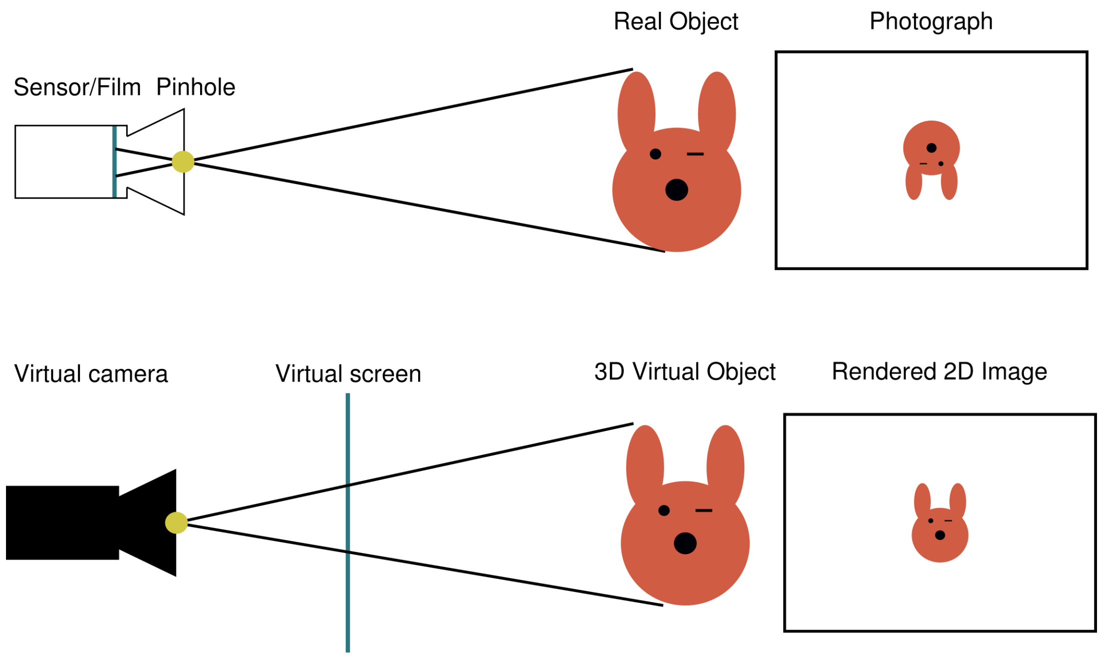

Figure 1.1. *The conceptual difference between the process of taking a photograph with a real camera (top) and rendering a 2D image of a virtual object (bottom) is the position of the sensor or film. In a pinhole camera model, light (black lines) passes through a pinhole (yellow dot) in both a real and virtual scene, but the sensor or film (short blue line) is located behind the pinhole whereas the virtual screen (long blue line) is situated in front of the virtual camera with respect to the scene. This setup makes the photograph (top right) flipped on both axis and the rendered 2D image (bottom right) appear as seen by the naked eye.*

pixels produces a perspectively correct projection of the 3D scene onto the virtual screen [49, pp. 4–15].

In the turn of the millennium, the rapid development of dedicated rendering hardware, namely graphics processing units (GPU), advanced the possibilities of real-time[1] rendering [4, p. 29]. Design decisions had to be made for good use of the limited hardware resources, and GPU rendering pipelines started to form into efficient fixed pipelines. To use the pipeline efficiently assumptions about the basic building blocks of rendering had to be made. The triangle was established as the *rendering primitive* of choice, which is highly prevalent in computer graphics to date. Rendering primitive refers to the lowest level geometric model from which the objects are constructed in a rendering pipeline.

Point cloud rendering

This thesis focuses on the various methods used for geometry processing in a ray tracing rendering pipeline for *point clouds*. Point clouds are a set of points in 3D space, which is mathematically defined as a set $P = \{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n\}$ with each $\boldsymbol{p}_i$ being a point in 3D space.

[1]In this thesis, *real-time* refers to rendering systems achieving at least 1 frame per second (fps) on consumer hardware.

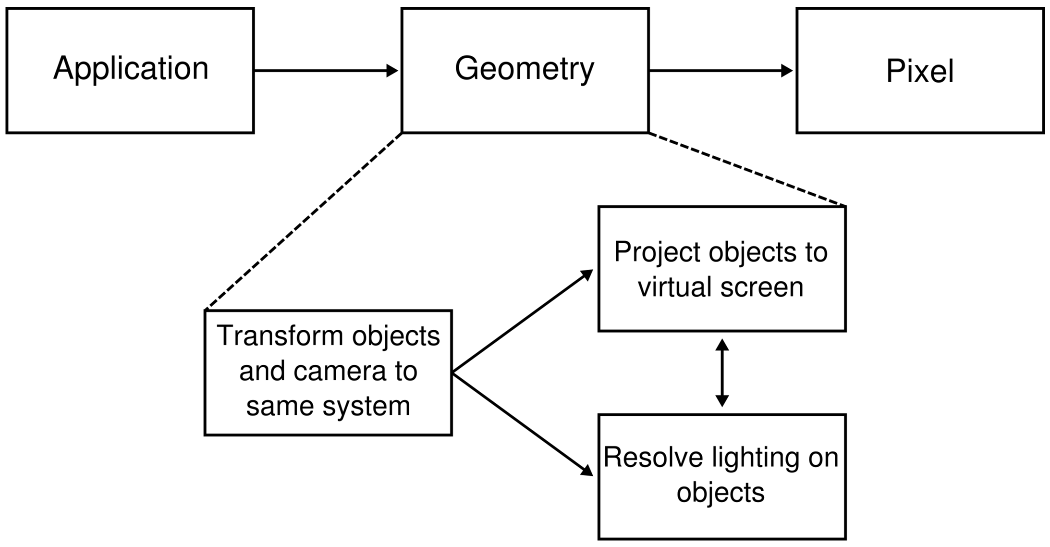
Application
Geometry
Pixel
Project objects to virtual screen
Transform objects and camera to same system
Resolve lighting on objects

Figure 1.2. *An overview of the rendering pipeline and the geometry stage in particular are depicted. In the geometry stage, objects in the scene, including the virtual camera, are first transformed to the same coordinate system. Then, either the lighting is resolved on the object surfaces or the objects are projected onto the virtual screen first.*

Each point may have additional information attached to it such as local point orientation and color. The mathematical properties of such a set are discussed in Chapter 3 and applied in Chapter 5.

The point cloud rendering research started in 1985 when Levoy and Whitted suggested using points as a basic building block for generating 2D images out of 3D virtual objects [40]. They argued that the increase in the complexity and size of virtual scene models would favour the use of points instead of triangles in rendering. Before this, triangles and other polygons arranged in a mesh structure were considered the main rendering primitive.

During the 2000s, several publications covering new point cloud ray tracing methods reported the growing interest in point-based rendering methods. It was argued that the growing complexity of object geometry was leading to triangle primitives being projected onto smaller areas than one pixel on the virtual screen, which was wasteful compared to using points [2, 55]. Furthermore, the availability of high detail point cloud models from 3D scanning devices afforded the possibility of high quality direct rendering of points without producing a connected set of triangles or other global *surface reconstructions* prior to rendering [63]. The subject of surface reconstruction and a survey of the different methods for efficient point cloud ray tracing are presented in Chapter 5.

Ray tracing

In 1980, Whitted improved the early version of ray casting into an alternative rendering method for rasterization known as *Whitted ray tracing* [65]. Instead of projecting all potentially visible models from the 3D scene onto the screen in rasterized graphics, in ray tracing the idea is to follow single light rays and their interactions in the virtual scene and use the information to deduce the color of each virtual screen pixel. Whitted's ray tracing algorithm only considered direct lighting (*shadow rays*), perfect reflections (*reflection rays*) and refractions (*refraction rays*). The ray tracing algorithm is discussed in Chapter 4.

The benefit of ray tracing, compared to traditional rasterizing, is the more realistic modelling of light interactions in a scene. Ray tracing has the potential to render images of virtual scenes that look photorealistic. Since Whitted, ray tracing algorithms have evolved to support more global effects of light. An advanced version of ray tracing, *path tracing*, approximates the complex paths of single light rays in a scene. Traditional Whitted ray tracing considers only single rays with predetermined ray paths where as path tracing extends the ray paths by *stochastically* generating new directions for the rays. This produces photorealistic effects like soft shadows, reflections and *global illumination* (GI). However, it is computationally impractical to follow the path of each individual light ray in a scene, thus only a subset of light rays are sampled and used as a representation of the contribution of all rays to the final image [4, pp. 443–445]. This is further discussed in Chapter 4. Furthermore, in Section 3.5, we discuss the mathematical theory of probability, distributions and sampling.

Even though the ray tracing technique has been available for more then half a century, the use of high quality ray tracing in a real-time rendering pipeline has become plausible only a few years ago [36, 56]. Real-time ray tracing pipelines can generally afford only a few samples per pixel (spp), thus the produced noisy rendered images have to be filtered in a suitable manner. The limited sample budget is due to the expensive nature of finding an intersection between a ray and a virtual scene consisting of up to hundreds of millions of rendering primitives [4, p. 818].

To ease the search for the closest or any intersection hierarchical data structures have been utilized [4, pp. 817–818]. These methods are usually referred to as *acceleration structures*. The main idea in any acceleration structure is to divide the virtual scene into relevant sub-sections which can be traversed. This is illustrated in Figure 1.3. The traversal phase consists of deciding whether it is possible that the ray has passed through a certain area of the scene or not. This means recursively sub-dividing the geometry into easily intersectable objects that encapsulate the finer detailed geometry and models. Usually these data structures lower the asymptotic complexity of intersection testing from $O(n)$ to $O(\log n)$ for each individual intersection test [4, p. 819].

Structure of the thesis

This thesis aims to answer the following research questions:

- Is it possible to ray trace point clouds directly in real-time without producing an explicit global surface reconstruction?

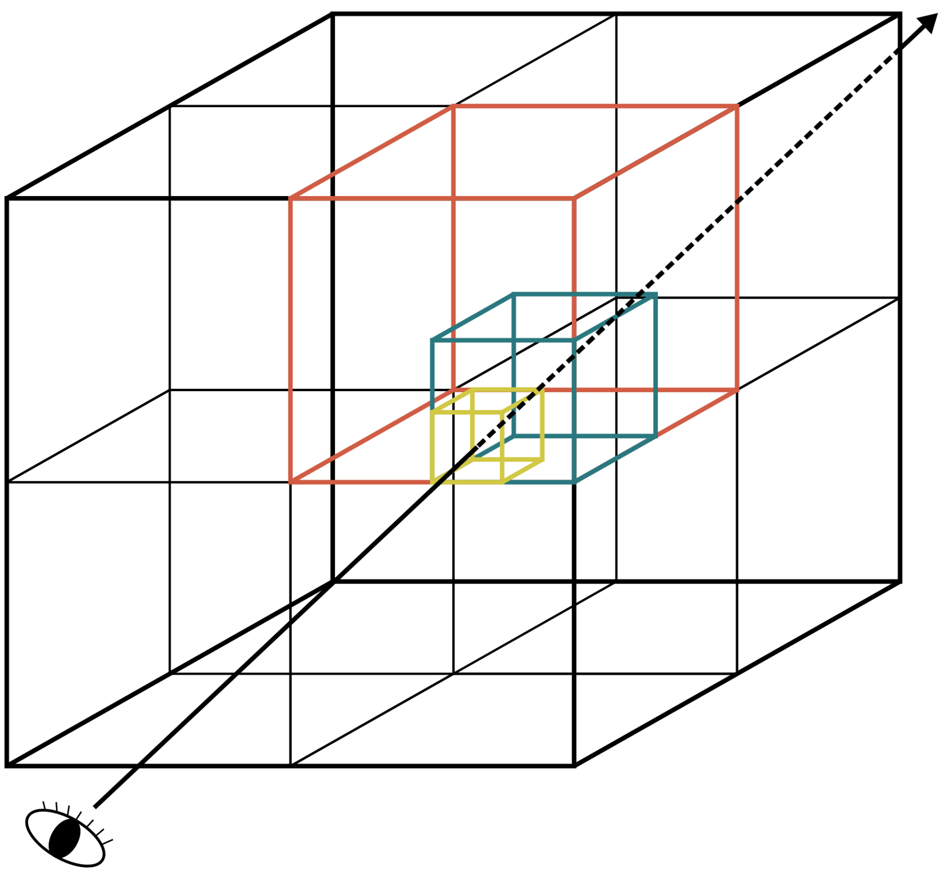

Figure 1.3. *The closest intersection between a viewing ray (solid line from the eye) and the face of an octree structure is depicted. An octree hierarchically sub-divides a subset of 3D space into eight equally sized cubes at multiple levels recursively. Intersection is first tested between the viewing ray and the top level (outer black cube) or level-0. If an intersection is found, then the next lower level (inner black cubes) or level-1 is tested for intersections, and the intersection testing is continued iteratively inside the intersected level-1 cube (red cube) until the lowest level of the octree structure is found (yellow cube).*

- What are the benefits (if any) of ray tracing point clouds directly?
- What are computationally the most efficient methods for point cloud ray tracing?

The methodologies used are surveying the scientific literature for methods on efficient point cloud ray tracing and analyzing the efficiency of the methods by asymptotic complexity analysis. Additionally this thesis gives a mathematical perspective to the subject of rendering point clouds in general.

We discuss and review the challenges in point cloud rendering in general and ray tracing point based models in specific in Chapter 2. Mathematical theory and notation used in this thesis are established in Chapter 3. Techniques for photorealistic and physically based rendering are reviewed in Chapter 4 with the main focus on ray tracing techniques. In Chapter 5, we survey methods for efficient ray tracing of point clouds and compare them to surface reconstruction methods with asymptotic complexity analysis. The results of the method comparison are summarized and discussed in Chapter 6. Finally, we conclude the thesis and discuss its limitations and possible future work in Chapter 7.

2 CHALLENGES IN REAL-TIME POINT CLOUD RENDERING

The area of point cloud rendering has been well researched [26, p. 1]. The challenge has been how to efficiently represent points as primitives and how to project them onto a monitor with high visual quality. Already in the early days of real-time computer graphics Levoy and Whitted argued that the rapid growth in model complexity would lead to computational challenges [40]. They suggested the use of points instead of polygons, such as triangles, as rendering primitives. Their argument was that triangles and other polygon primitives would be projected onto sub-pixel areas on the monitor. In these cases a triangle primitive could be just as well represented by a single point. Since the publication by Levoy and Whitted, various methods for making point cloud rendering efficient have been published, which are discussed and surveyed in Chapter 5.

One key challenge in point cloud rendering compared to polygonal model rendering is that no explicit surface representation exists in point based geometry. This means that the surface represented by points has to be deduced by the point locations and possibly other attributes so that it can be visualized. Surface reconstruction methods for point based models have been widely proposed to tackle the problem [11]. However, most of the reconstruction methods surveyed by Berger et al. are not aimed at real-time reconstruction and thus are not necessarily applicable to real-time rendering.

Research in particle-based simulation has been actively interested in generating real-time visualizations of their simulations [52]. A simulation particle can be abstracted as a point with shape, volume, and acting forces, thus point based rendering methods can be utilized for visualization purposes. The challenge is similar to point cloud rendering in general: how can simulations with millions or billions of particles be rendered or visualized efficiently. Recently, efficient solutions to the problem of increasing complexity have been applied, namely hierarchical *level-of-detail* (LOD) systems and acceleration structures, as well as some approximate solutions to improve realistic lighting and the perception of depth like ambient occlusion (AO) [52]. However, these methods have been developed earlier in the field of computer graphics and they are further discussed in the context of physically based rendering (PBR) and efficient point cloud ray tracing in Chapters 4 and 5, respectively.

Compared to PBR, particle simulation visualization is more functional, i.e., it is more interested in generating illustrative renderings that highlight some parts of a process in molecular or atomic level than producing accurate color and lighting effects on point cloud models. In this thesis the particle-based visualization methods are not considered further and the reader is referred to [52] for a state-of-the-art survey on the subject.

2.1 Intersection testing

A raw point cloud model doesn't explicitly have a surface to intersect because it is a collection of points without connectivity information. Wald et al. [63] pointed out that because there is a finite number of points and an infinite number of locations in any subset of 3D space, the probability of a randomly sampled ray (line) to pass through a point in 3D space is zero. This is further discussed in Section 3.5.

There are several approaches to making intersection possible between a ray and a point cloud. Rays can be expanded from mere lines into objects with volume or, conversely, points can be enlarged to have surface area [63]. Effectively, this means using geometrical objects like cylinders [55] to represent the ray as an object with volume or expand points into volumes or surfaces to generate, e.g., discs [68].

An intersection between a ray and an object surface can be solved analytically in a closed form solution or iteratively with numerical methods. Numerical and iterative methods are discussed in Section 3.6. The analytical methods are usually restricted to surfaces defined by polynomials of 4th order or lower because there exists no general closed form solutions to the roots expressed by the parameters of the high order polynomial [53]. Intersection testing as a solution to a multivariate polynomial equation is discussed in Section 3.4.

Iterative intersection methods for point clouds, such as [14, 63], start with an initial guess of the intersection usually generated by a simple bounding representation of the surface patch to be intersected. The initial guess is then fed back to the algorithm to produce a more refined approximation of the intersection. When some criterion for the accuracy of the approximation is achieved, the iteration is stopped and the current intersection is reported. The usefulness of an iterative method in a real-time application is highly dependent on the iteration count. These algorithms usually guarantee some maximum iteration count or exit after some amount of iterations to stay in a predefined timing budget to achieve real-time frame rates.

2.2 Surface definition

A point cloud surface can be defined in many ways (Chapter 5). The problem of reconstructing a surface out of point-sampled geometry is analogous to the surface definition itself. As discussed previously, surface reconstruction has been a time consuming process and real-time applications haven't been possible [11]. However, some real-time surface reconstruction methods have been proposed [42, 46, 47], which utilize the readily available red-green-blue-depth (RGB-D) imaging capabilities of modern handheld devices and medical imagery. However, these methods only produce the reconstruction, which still leaves the rendering to be done. Many surface reconstruction methods use approximation and estimation methods, which are further discussed in Section 3.6.

Surface definition is tightly linked with intersection testing because the defined surface has to be intersected when ray tracing. The traditional surface representation for real-time rendering of point clouds has been the splatting technique [54, 68], in which points are represented by flat linear surfaces like discs, which can be elliptically shaped. Furthermore, splats can be oriented by normal directions or be aligned with the viewing direction, where the trade-off is between image quality and rendering speed. More advanced splatting techniques define the size of elliptical discs by the local density [51] or curvature [41] of the point cloud to produce a more faithful representation of the sampled surface. Splats produce a piece-wise continuous and linear definition of the surface. Furthermore, splats are simple to intersect (see Section 3.4) making them a viable representation for real-time ray tracing of point clouds.

2.3 Advanced lighting effects

After finding an intersection between a ray and a surface, the lighting and color information at the intersection point have to be evaluated. Rasterization methods traditionally rely on approximate shading to produce diffuse and specular effects as well as shadows on surface primitives projected on to the screen, which produces various amounts of approximation error [4, p. 433].

Instead of using the traditional methods of rasterizing point clouds, it is possible to use more sophisticated rendering techniques to produce realistic lighting effects on point cloud models. The traditional Whitted ray tracing considers the effect of reflecting and refracting light. Furthermore, path tracing is an all purpose method of recursively producing effects of real life GI in various lighting conditions. These kind of effects are important in photorealistic rendering and PBR, which are commonly used in the movie industry [13, 17].

Generating advanced lighting effects with ray tracing is costly [4, p. 445]. Efficient methods for ray tracing and path tracing have been well researched [62] and many publications in point cloud ray tracing have also been published (Section 5.3). Nowadays, real-time ray tracing is mostly limited to a strict ray budget which means that only a few rays can be traced through each virtual pixel, and furthermore, the scattered light at each ray-surface intersection can only be sampled a limited number of times [36, 56]. It is important that the rays that will be sampled are used efficiently, for example, by sampling directions with the most impact on the final color of a pixel. This equates to taking representative samples of an underlying probability distribution and approximating the mean of the distribution based on a weighted sum of the samples. See Section 3.5 for further details on distributions and sampling.

3 MATHEMATICAL THEORY

In this chapter we define the necessary mathematical notations and properties needed in point cloud ray tracing and path tracing. We present the relevant results from set theory, vector and matrix calculus, probability and statistical theory, approximation and estimation theory, and polynomial functions and root finding. These are later applied in Chapters 4 and 5 to rendering in general and efficient ray tracing of point clouds in specific.

3.1 Set theory

A set is a collection of elements, namely

$$A = \{a_1, a_2, \ldots, a_n\},$$

where a_i, $i = 1, \ldots, n$ are elements. Set A is said to be *finite* if $n \in \mathbb{N}$. If $n \to \infty$, then A is said to be *infinite* and by construction *numerable*. We can also define a set consisting of elements satisfying some property:

$$A = \{x : F(x)\}, \tag{3.1}$$

where A is the set to be defined and F is a formula which is satisfied by an element x of the set. As stated in [60, p. 105] this notation means logically $x \in A \Leftrightarrow F(x)$, where the binary relations $\in$ and $\notin$ indicate the membership and non-membership of an element to a set, respectively.

Basic set relations include equality ($=$) and subset ($\subseteq$) which are defined in the following way:

$$A = B \Leftrightarrow (x \in A \Leftrightarrow x \in B) \tag{3.2}$$

$$A \subseteq B \Leftrightarrow (x \in A \Rightarrow x \in B). \tag{3.3}$$

3.2 Metric spaces

A metric space defines the notion of "distance" for a set of elements. A metric is defined in the following way [20, p. 27]:

Definition 3.1. A *metric* is a mapping $d : E \times E \to \mathbb{R}$ on the elements of a set E with the following properties for all $a, b, c \in E$:

(i) $d(a,b) \geq 0$

(ii) $d(a,b) = 0 \Leftrightarrow a = b$

(iii) $d(a,b) = d(b,a)$

(iv) $d(a,c) \leq d(a,b) + d(b,c)$

The metric attached to the point cloud set P defines the abstract concept of distance which will be further developed into a norm in a vector space.

3.3 Vector spaces

We define a vector space in n-dimensional *Euclidian space* $\mathbb{R}^n$, following the definitions in [43, pp. 11–17].

Definition 3.2. A non-empty subset $\mathbb{V} \subseteq \mathbb{R}^n$ together with addition and scalar multiplication is called a *vector space* if for all elements $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c} \in \mathbb{V}$ and scalars $\alpha, \beta \in \mathbb{R}$ the following hold:

(i) $(\boldsymbol{a} + \boldsymbol{b}) + \boldsymbol{c} = \boldsymbol{a} + (\boldsymbol{b} + \boldsymbol{c})$

(ii) $\alpha(\beta \boldsymbol{a}) = (\alpha\beta)\boldsymbol{a}$

(iii) $\boldsymbol{a} + \boldsymbol{b} = \boldsymbol{b} + \boldsymbol{a}$

(iv) $\alpha(\boldsymbol{a} + \boldsymbol{b}) = \alpha\boldsymbol{a} + \alpha\boldsymbol{b}$

(v) $\boldsymbol{a}(\alpha + \beta) = \boldsymbol{a}\alpha + \boldsymbol{a}\beta$

(vi) $\exists\ \boldsymbol{0} \in \mathbb{R}^n : \boldsymbol{a} + \boldsymbol{0} = \boldsymbol{a}$

(vii) $\exists\ -\boldsymbol{a} \in \mathbb{R}^n : \boldsymbol{a} + (-\boldsymbol{a}) = \boldsymbol{0}$

The term *vector* is adopted to mean the element of a vector space i.e. an element of a set which completes definition 3.2 with appropriate addition and scalar multiplication.

Remark 3.3. The set $\mathbb{R}^n$ with element-wise addition and scalar multiplication is a vector space.

Definition 3.2 is an abstract construction of a vector space over the field of real numbers. It is convenient to introduce a way of constructing a vector space from a basis. The concept of linear dependence is needed.

Definition 3.4. The set $\{\boldsymbol{a}_i\}_{i=1,\dots,m} \subset \mathbb{V}$ is *linearly dependent* if there exists $\alpha_i \in \mathbb{R}, i = 1, \dots, m$ and at least one $\alpha_i \neq 0$ s.t.

$$\sum_{i=1}^{m} \alpha_i \boldsymbol{a}_i = \boldsymbol{0}. \tag{3.4}$$

Conversely, if the equation (3.4) holds only with $\alpha_i = 0$ for all i, then we say that the set of vectors is *linearly independent*.

Definition 3.5. The set $\mathcal{B} = \{\boldsymbol{b}_i\}_{i=1,\dots,m}$ is a basis of a vector space $\mathbb{V} \supset \mathcal{B}$ if for all $\boldsymbol{v} \in \mathbb{V}$ there exists $\alpha_i \in \mathbb{R}$ such that

$$\boldsymbol{v} = \sum_{i=1}^{m} \alpha_i \boldsymbol{b}_i \tag{3.5}$$

and $\mathcal{B}$ is linearly independent.

By definition we require that the basis of a given vector space is linearly independent and thus the zero element cannot belong to the basis of any vector space.

The basis of a specific vector space $\mathbb{V}$ can be denoted $\mathcal{B}_{\mathbb{V}}$ for clarity. The representation of equation (3.5) is unique in a given basis, i.e., the coefficients α_i are unique for a given element of $\mathbb{V}$. We omit the full proof but refer to [27, pp. 193–194] for a justification. Finally we show that the standard basis of the vector space $\mathbb{R}^n$ is truly a basis. We use the notation from [27, p. 195], particularly the Kronecker delta:

$$\delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise.} \end{cases}$$

Theorem 3.6. *The set $\{\boldsymbol{e}_i\}_{i=1,\dots,n}$, $\boldsymbol{e}_i = (\delta_{1i}, \dots, \delta_{ni})$, is a basis of $\mathbb{R}^n$.*

Proof. Clearly $\{\boldsymbol{e}_i\}_{i=1,\dots,n} \subset \mathbb{R}^n$. Let $\boldsymbol{v} = (v_1, \dots, v_n) \in \mathbb{R}^n$. Now we have $\sum_{i=1}^{n} v_i \boldsymbol{e}_i = \sum_{i=1}^{n} v_i(\delta_{1i}, \dots, \delta_{ni}) = (v_1, 0, \dots, 0) + \cdots + (0, \dots, v_n) = (v_1, \dots, v_n) = \boldsymbol{v}$ so we have coefficients to satisfy Equation (3.5). Suppose $\{\boldsymbol{e}_i\}_{i=1,\dots,n}$ is linearly dependent so we have $\sum_{i=1}^{m} \alpha_i \boldsymbol{e}_i = \boldsymbol{0}$ and there exists i s.t. $\alpha_i \neq 0$ for some $i = j = 1, \dots, n$. Now $\sum_{i=1}^{m} \alpha_i \boldsymbol{e}_i = (\alpha_1, 0, \dots, 0) + \cdots + (0, \dots, \alpha_j, \dots, 0) + \cdots + (0, \dots, \alpha_n) = (\alpha_1, \dots, \alpha_j, \dots, \alpha_n)$. Because $\alpha_j \neq 0$ also $(\alpha_1, \dots, \alpha_j, \dots, \alpha_n) \neq \boldsymbol{0}$ which contradicts linear dependency. Thus $\{\boldsymbol{e}_i\}_{i=1,\dots,n}$ is linearly independent. $\square$

We can express the whole $\mathbb{R}^n$ vector space with the *linear combination* of the basis $\{\boldsymbol{e}_i\}_{i=1,\dots,n}$ which we denote with $\mathcal{E}^n$.

Finally a definition for a norm in a vector space is introduced. With this we can define a normed vector space which is the space that is used as the starting point for vector analysis. It can be shown that a norm is a metric and furthermore a normed vector space is a metric space. We use a definition of norm adapted from [20, p. 88] and [43, p. 22].

Definition 3.7. Let $\mathbb{V} \subseteq \mathbb{R}^n$ be a vector space. The mapping $\|\cdot\| : \mathbb{V} \to \mathbb{R}$ is a *norm* if for all $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{V}$ and $\alpha \in \mathbb{R}$ the following hold:

(i) $\|\boldsymbol{a}\| \geq 0$

(ii) $\|\boldsymbol{a}\| = 0 \Leftrightarrow \boldsymbol{a} = \boldsymbol{0}$

(iii) $\|\alpha\boldsymbol{a}\| = |\alpha|\|\boldsymbol{a}\|$

(iv) $\|\boldsymbol{a}+\boldsymbol{b}\| \leq \|\boldsymbol{a}\| + \|\boldsymbol{b}\|.$

A vector space with a norm is called a *normed vector space*. The vector space $\mathbb{R}^n$ together with the Euclidian norm

$$\|\boldsymbol{x}\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}, \tag{3.6}$$

is a normed vector space. The Euclidian norm satisfies the properties 3.7 (i), (ii) and (iii), and the triangle inequality property 3.7 (iv) follows from the *First Minkowski's inequality* [19, p. 98]. From now on, we will use the general norm notation $\|\cdot\|$ to refer to the Euclidian norm in Equation (3.6) and state explicitly if we are using some other norm.

We define the *dot product* [37, pp. 408–410].

Definition 3.8. The operator $\cdot : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ defined as

$$\cdot(\boldsymbol{u},\boldsymbol{v}) := \boldsymbol{u}\cdot\boldsymbol{v} = \sum_{i=1}^{n} u_i v_i,$$

where $\boldsymbol{u} = (u_1,\ldots,u_n) \in \mathbb{R}^n$ and $\boldsymbol{v} = (v_1,\ldots,v_n) \in \mathbb{R}^n$, is called the *dot product*.

Even though the dot product is defined in n-dimensional vector space, it has an interesting property in three-dimensional space. More accurately the property is on the two-dimensional plane which is spanned by the vectors $\boldsymbol{u}$ and $\boldsymbol{v}$. The smallest angle $\alpha \in [0,\pi]$ between two vectors in $\mathbb{R}^n$ on the plane that the vectors span is linked to the dot product by the identity

$$\boldsymbol{u}\cdot\boldsymbol{v} = \|\boldsymbol{u}\|\,\|\boldsymbol{v}\|\cos\alpha. \tag{3.7}$$

Sometimes the definition of angle is given as a modified version of (3.7) in the form

$$\cos\alpha = \frac{\boldsymbol{u}\cdot\boldsymbol{v}}{\|\boldsymbol{u}\|\,\|\boldsymbol{v}\|}, \tag{3.8}$$

which gives a standard notion of angle in higher dimensions. If for $\boldsymbol{u},\boldsymbol{v} \neq \boldsymbol{0}$ and $\boldsymbol{u}\cdot\boldsymbol{v} = 0$, then $\cos\alpha = \frac{\pi}{2}$ and we say that $\boldsymbol{u}$ and $\boldsymbol{v}$ are *orthogonal* or perpendicular to each other, denoted $\boldsymbol{u} \perp \boldsymbol{v}$. Furthermore, if additionally $\|\boldsymbol{u}\| = \|\boldsymbol{v}\| = 1$, then we call them *orthonormal*. Now we can show that the basis described in Theorem 3.6 for $n = 3$, namely $\mathcal{E}^3$, is an orthonormal basis in $\mathbb{R}^3$.

Theorem 3.9. *$\mathcal{E}^3 \subset \mathbb{R}^3$ is an orthonormal basis.*

Proof. Equation (3.5) establishes that $\mathcal{E}^3$ is a basis, $\boldsymbol{e}_1\cdot\boldsymbol{e}_2 = \boldsymbol{e}_2\cdot\boldsymbol{e}_3 = \boldsymbol{e}_3\cdot\boldsymbol{e}_1 = 0$ and

$\|\boldsymbol{e}_1\| = \|\boldsymbol{e}_2\| = \|\boldsymbol{e}_3\|$. □

The elements of $\mathcal{E}^3$ define the canonical x, y and z axis in a *Cartesian Coordinate system*.

3.3.1 Vector analysis

We define the *gradient* of a multi-variable scalar valued function $f : \mathbb{R}^n \to \mathbb{R}$ denoted $f(x_1, \ldots, x_n) := f(\boldsymbol{x})$ in the domain of Euclidian space $\mathbb{R}^n$ [37, pp. 446–450].

Definition 3.10. The *gradient* of a scalar valued function $f : \mathbb{R}^n \to \mathbb{R}$ is

$$\nabla f(\boldsymbol{x}) = \left(\frac{\partial f(\boldsymbol{x})}{\partial x_1}, \ldots, \frac{\partial f(\boldsymbol{x})}{\partial x_n} \right),$$

where $\frac{\partial f(\boldsymbol{x})}{\partial x_i}$ is the *partial derivative* with respect to variable x_i for any i. This is evaluated by differentiating $f(\boldsymbol{x})$ with respect to x_i, while treating the other variables $x_j,\ j \neq i$ as constants in the differentiation. The gradient $\nabla f(\boldsymbol{x})$ at point $\boldsymbol{x} \in \mathbb{R}^n$ is a vector in the domain of f that points to the direction of greatest increase in the value of $f(\boldsymbol{x})$ at a small neighborhood of $\boldsymbol{x}$.

For the domains $\mathbb{R}^3$ and $\mathbb{R}^2$ with the Cartesian coordinates x, y and z as their basis, the gradients are

$$\begin{aligned} \nabla f(x,y,z) &= \left(\frac{\partial f(x,y,z)}{\partial x}, \frac{\partial f(x,y,z)}{\partial y}, \frac{\partial f(x,y,z)}{\partial z} \right) \\ \nabla f(x,y) &= \left(\frac{\partial f(x,y)}{\partial x}, \frac{\partial f(x,y)}{\partial y} \right), \end{aligned}$$

respectively.

3.4 Polynomial functions

Polynomial functions can be categorized according to their degree and number of variables. Informally the degree of a polynomial describes the expressiveness of the function, i.e., how many points are needed to represent the polynomial uniquely. For example a polynomial function $p(x) = ax^3 + bx^2 + cx + d$ is of degree 3 and is defined uniquely by 4 points.

Another important categorization for polynomials and functions is the form they are represented in. The *explicit form* expresses a function with polynomial variables and their coefficients, and it can be geometrically interpreted by setting the variables and the function value as coordinates in space. The *parametric form* makes the space and coordinates explicit by setting each coordinate dependent on some polynomial with bounded variables and their coefficients. Finally the *implicit form* defines a set of points by setting a condition that an implicit polynomial has to satisfy. Examples of these forms are presented later.

A single variable real-valued polynomial function $p : \mathbb{R} \to \mathbb{R}$ is of the following explicit form [10, p. 1]:

$$p(x) = \sum_{i=0}^{n} a_i x^i,$$

where $a_i \in \mathbb{R}$ for $i = 1, \ldots, n$. The degree of p is n, if $a_n \neq 0$. The roots of a single variable polynomial cannot be solved analytically if the degree of the polynomial is larger than 4 [10, p. 126].

In this section we consider real valued polynomials with up to two variables in explicit form and three in implicit form. These polynomials are of special interest because they define a surface in $\mathbb{R}^3$. Surfaces can be represented geometrically in $\mathbb{R}^3$, thus they have an application in computer graphics. Finding the intersections and distances between lines and surfaces are used for generating projected images in $\mathbb{R}^2$ of surfaces in $\mathbb{R}^3$.

3.4.1 Multivariate polynomials

A general formulation of a multivariate real valued polynomial can be given as [50, p. 148]

$$p(x_1, \ldots, x_n) = \sum_{k_i \leq m} a_{k_1 \ldots k_n} x_1^{k_1} \ldots x_n^{k_n},$$

where the sum is over all possible combinations of $k_i \leq m \in \mathbb{N}$, $i = 1, \ldots, n$ and m is the highest degree of a single *monomial* term. Restricting the number of variables up to two yields bivariate polynomials of the form

$$p(x, y) = \sum_{\alpha, \beta \leq m} a_{\alpha\beta} x^\alpha y^\beta, \tag{3.9}$$

where $\alpha, \beta \in \mathbb{N}$.

Setting $z = p(x, y)$ in a Cartesian coordinate system yields a definition for a surface in $\mathbb{R}^3$. For example, take $z = x^2 + y^2 + xy$ and $x, y \in [-1, 1]$, which is presented in Figure 3.1.

Parametric form

The parametric form of a polynomial function defines the geometric interpretation of the function explicitly. Each coordinate parameter is expressed by variables that connect the parameters. For a two variable polynomial in $\mathbb{R}^3$ the form is expressed in the following way:

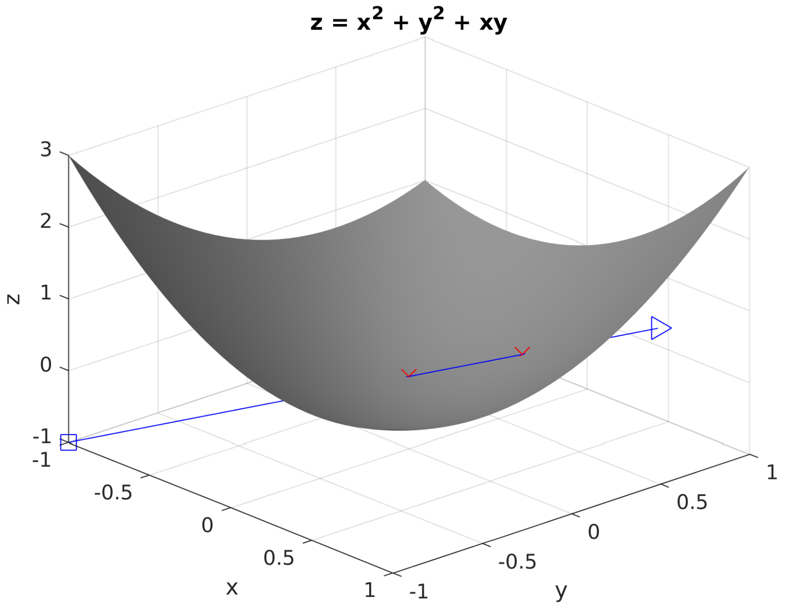

Figure 3.1. *A polynomial surface intersected by a ray in $\mathbb{R}^3$. The ray (blue line) intersects the polynomial surface at two points (red crosses) which are in bounds of the ray origin (blue square) and end point (blue triangle).*

$$\boldsymbol{p}(u,v) = \begin{bmatrix} p_x(u,v) \\ p_y(u,v) \\ p_z(u,v) \end{bmatrix} = \begin{bmatrix} \sum_{\alpha,\beta \le m_x} a_{\alpha\beta} u^\alpha v^\beta \\ \sum_{\alpha,\beta \le m_y} b_{\alpha\beta} u^\alpha v^\beta \\ \sum_{\alpha,\beta \le m_z} c_{\alpha\beta} u^\alpha v^\beta \end{bmatrix},$$

where $a_{\alpha\beta}, b_{\alpha\beta}, c_{\alpha\beta} \in \mathbb{R}$, $u, v \in \mathbb{R}$ and $m_x, m_y, m_z \in \mathbb{N}$ are the degrees of the polynomials p_x, p_y, p_z, respectively.

The parametric form of a *line segment* in $\mathbb{R}^3$ is

$$[r_x(t), r_y(t), r_z(t)]^T := \boldsymbol{r}(t) = \boldsymbol{o} + \boldsymbol{d}t = [o_x + d_x t,\ o_y + d_y t,\ o_z + d_z t]^T, \tag{3.10}$$

where $t \in [a,b] \subset \mathbb{R}$ and $\boldsymbol{o}, \boldsymbol{d} \in \mathbb{R}^3$ are the starting point and direction of the line segment, respectively. This is used as the abstraction for a *ray* in the rest of the work, thus it is referred to as a ray from now on and the interval is denoted $[a,b] := [t_{min}, t_{max}]$.

Implicit form

Setting a polynomial to fulfill a condition or equation defines the implicit form of a polynomial function. For surface definitions in $\mathbb{R}^3$ we restrict ourselves to three variable polynomials with the implicit function $p : \mathbb{R}^3 \to \mathbb{R}$ with the surface points satisfying the zero-level function, namely

$$p(\boldsymbol{x}) = p(x,y,z) \overset{(3.9)}{=} \sum_{\alpha,\beta,\gamma \leq m} a_{\alpha\beta\gamma} x^{\alpha} y^{\beta} z^{\gamma} = 0,$$

where $\boldsymbol{x} = [x, y, z] \in \mathbb{R}^3$ defines a surface in $\mathbb{R}^3$. Transforming a polynomial from explicit form to implicit form is straight forward but the converse is necessarily not.

Let us consider the implicit form of a plane $s \in \mathbb{R}^3$. For a plane with a vector $\boldsymbol{n}$ orthogonal to the plane and a point $\boldsymbol{p}$ on the plane, the implicit function $s(\boldsymbol{x})$ defining all the points $\boldsymbol{x}$ on the plane is given by

$$s(\boldsymbol{x}) = \boldsymbol{n} \cdot (\boldsymbol{x} - \boldsymbol{p}) = 0 \Rightarrow \boldsymbol{x} \in S. \tag{3.11}$$

By taking the ray $\boldsymbol{r}(t)$ from Equation (3.10), evaluating $s(\boldsymbol{r}(t))$ and solving for t we get

$$t_{\times} = \frac{\boldsymbol{n} \cdot (\boldsymbol{p} - \boldsymbol{o})}{\boldsymbol{n} \cdot \boldsymbol{d}} \wedge t_{\times} \in [t_{min}, t_{max}] \subset \mathbb{R} \Rightarrow \boldsymbol{r}(t_{\times}) \in s, \tag{3.12}$$

where $t_{\times}$ is the value of t at the intersection. If $\boldsymbol{n} \cdot \boldsymbol{d} = 0$, then the ray is parallel to s and there is no intersection between the ray and the plane, i.e., $t_{\times} \notin \mathbb{R}$.

Equation (3.12) is a slightly different derivation than in [4, p. 966], where $t_{\times} = (-d - \boldsymbol{n} \cdot \boldsymbol{o})/(\boldsymbol{n} \cdot \boldsymbol{d})$. However, by setting $d = -(\boldsymbol{n} \cdot \boldsymbol{p})$ they define the same plane and the equations are equivalent.

3.4.2 Examples of intersections

We present two examples of ray-surface intersections important for point cloud ray tracing. These include intersections with second order polynomial surfaces and spheres.

Polynomial surface patch

Let's examine the intersection between a ray $\boldsymbol{r}(t)$ and a surface s defined by a second order polynomial of a form $s(x,y) = z = a_1 x^2 + a_2 y^2 + a_3 xy + a_4$, where $a_1, a_2, a_3, a_4 \in \mathbb{R}$, $x \in [x_{min}, x_{max}]$ and $y \in [y_{min}, y_{max}]$. Place the ray parameters from Equation (3.10) to x, y and z variables to obtain

$$o_z + d_z t = a_1(o_x + d_x t)^2 + a_2(o_y + d_y t)^2 + a_3(o_x + d_x t)(o_y + d_y t) + a_4.$$

Solving for t gives us two roots

$$t = \frac{-3d_x o_y + d_z - 3d_y o_x - 2a_1 d_x o_x - 2a_2 d_y o_y \pm \sqrt{D}}{2(a_1 d_x^2 + 3d_x d_y + a_2 d_y^2)}, \tag{3.13}$$

where

$$\begin{aligned}D =& d_z^2 + 9d_x^2 o_y^2 + 9d_y^2 o_x^2 - 12a_4 d_x d_y + 12 d_x d_y o_z - 6d_x d_z o_y - 6d_y d_z o_x - 4a_1 a_4 d_x^2 \\ & - 4a_2 a_4 d_y^2 + 4a_1 d_x^2 o_z + 4a_2 d_y^2 o_z - 18 d_x d_y o_x o_y - 4a_1 a_2 d_x^2 o_y^2 - 4a_1 a_2 d_y^2 o_x^2 \\ & - 4a_1 d_x d_z o_x - 4a_2 d_y d_z o_y + 8a_1 a_2 d_x d_y o_x o_y.\end{aligned}$$

The ray and the surface patch intersect, if $t \in [t_{min}, t_{max}] \subset \mathbb{R}$, $o_x + d_x t \in [x_{min}, x_{max}]$ and $o_y + d_y t \in [y_{min}, y_{max}]$.

For example, take the surface $z = x^2 + y^2 + xy$, $x, y \in [-11]$ and $\boldsymbol{r}(t) = [-1, -1, -1]^T + \frac{1}{\sqrt{3}}[1, 1, 1]^T t$. By Equation (3.13) we get $t \approx 1.73$ or $t \approx 2.31$, and the intersection points $[0, 0, 0]^T$ and $[\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}]^T$, which can be seen in Figure 3.1.

Sphere

Let a sphere be defined by its implicit function $f(\boldsymbol{x}) = \|\boldsymbol{x} - \boldsymbol{c}\| - r^2 = 0$, where $\boldsymbol{c} \in \mathbb{R}^3$ is the sphere center and r its radius. Now by setting $f(\boldsymbol{r}(t)) = 0$ and solving for t we find the possible intersections

$$t_\times = -\boldsymbol{d} \cdot (\boldsymbol{o} - \boldsymbol{c}) \pm \sqrt{D}, \tag{3.14}$$

where $D = (\boldsymbol{d} \cdot (\boldsymbol{o} - \boldsymbol{c}))^2 - \|\boldsymbol{o} - \boldsymbol{c}\|^2 - r^2$ [3, p. 739]. From the equation we see that if $D < 0$, $D = 0$, or $D > +$ we have 0, 1 or 2 intersections, respectively. In the double intersection case, we can choose the smaller $t_\times$ as the closest intersection.

3.5 Probability and statistical theory

We follow the definition of probability from [12, pp. 329–344]. For a random variable $\underline{x}$, the set of all possible outcomes Ω is called the *sample space*. The probability of $\underline{x}$ exhibiting an *outcome* $X \subseteq \Omega$ is measured by the probability $\mathcal{P}(\underline{x} = X) := \mathcal{P}(X)$, where $\mathcal{P}$ has the following properties:

(i) $\mathcal{P}(X) \geq 0$

(ii) $\mathcal{P}(\Omega) = 1$

(iii) $\mathcal{P}\left(\bigcup\limits_{i=1}^{\infty} X_i\right) = \sum\limits_{i=1}^{\infty}(\mathcal{P}(X_i))$,

where $X \subseteq \Omega$ is any outcome and $X_i \subseteq \Omega$ are *disjoint* sets, i.e., $X_i \cap X_j = \emptyset,\ i \neq j$.

3.5.1 Distributions

A *probability density function* (PDF) expresses the probability density for the different outcomes of a random variable $\underline{x}$. A *Gaussian* function $\mathcal{G} : \mathbb{R} \to \mathbb{R}$ is the continuous PDF of a normally distributed random variable $\underline{x} \sim \mathcal{N}(\mu, \sigma^2)$ defined as

$$\mathcal{G}(t) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(t-\mu)^2}{2\sigma^2}}, \tag{3.15}$$

where t is the outcome of $\underline{x}$ and $\mu, \sigma^2 \in \mathbb{R}$ are the *mean* and *variance* of the normal distribution, respectively [21, pp. 82–99]. To derive a probability measure for $\underline{x}$, we integrate $\mathcal{G}$ over the interval $[-\infty, x]$ to obtain

$$\mathcal{P}(\underline{x} < x) = \int_{-\infty}^{x} \mathcal{G}(t) = \frac{1}{\sqrt{2\pi\sigma^2}} \int_{-\infty}^{x} \exp -\frac{(t-\mu)^2}{2\sigma^2} dt, \tag{3.16}$$

where $\mathcal{P}(\underline{x} < x)$ is the probability of $\underline{x}$ having an outcome smaller than x [48, p. 103]. Equation 3.16 satisfies the properties of a probability measure.

A continuous *joint bivariate density function* $f_{X,Y}(x, y)$ [21, p. 211] for random variables X and Y is defined as

$$\mathcal{P}((X, Y) \in B) = \int\int_{B} f_{X,Y}(x, y) dy dx, \tag{3.17}$$

where $\mathcal{P}((X, Y) \in B)$ is the probability of (X, Y) being on some domain $B \in \mathbb{R}^2$. By taking $B := \Omega$ to be the unit sphere at origin, namely points (x, y) satisfying $x^2 + y^2 = 1$ and denoting $(X, Y) := \boldsymbol{l}$ and $dydx := d\boldsymbol{l}$ we obtain the density function $f(\boldsymbol{l})$ over the unit sphere

$$\mathcal{P}(\boldsymbol{l} \in \Omega) = \int_{\Omega} f(\boldsymbol{l}) d\boldsymbol{l}, \tag{3.18}$$

where $\int_{\Omega}$ stands for the double integral over the unit sphere.

3.5.2 Importance sampling

Given a function $f : \mathbb{R}^2 \to \mathbb{R}$, $\boldsymbol{x} \in \mathbb{R}^2$, and a probability density function $p : \mathbb{R}^2 \to \mathbb{R}$, we can estimate the integral over the domain $\Omega \subseteq \mathbb{R}^2$ by

$$\int_{\Omega} f(\boldsymbol{x}) d\boldsymbol{x} = \boldsymbol{E}\left\{\frac{f(\boldsymbol{x})}{p(\boldsymbol{x})}\right\} \approx \frac{1}{n} \sum_{i=1}^{n} \frac{f(X_i)}{p(X_i)}, \tag{3.19}$$

where $\boldsymbol{E}$ is the expected value and X_i are uncorrelated samples drawn from the distribution of the density function p. The estimate in Equation (3.19) has a variance of $\frac{\sigma^2}{N}$, where σ^2 is the variance of $\dfrac{f(\boldsymbol{x})}{p(\boldsymbol{x})}$ [30, p. 854]. We use this result in Section 4.3.1 to derive an importance sampled version of the rendering equation.

3.6 Approximation and estimation

This section introduces the approximation and estimation methods used in Chapter 5. Interpolation methods, weighted average, least squares estimates, and iterative algorithms for numeric estimation are presented.

3.6.1 Linear interpolation

Let $f : \mathbb{R} \to \mathbb{R}$ and $x \in [x_1, x_2] \subseteq \mathbb{R}$. For two evaluation points $(x_1, f(x_1))$ and $(x_2, f(x_2))$, the *Lagrange formula* defines a first order polynomial

$$p_1(x) = \frac{(x_2 - x)f(x_1) + (x - x_1)f(x_2)}{x_2 - x_1}, \tag{3.20}$$

which approximates the *linearly interpolated* value of $f(x) \approx p_1(x)$ [8, p. 134].

Bilinear interpolation extends the linear interpolant to $\mathbb{R}^2$. Let $g : \mathbb{R}^2 \to \mathbb{R}$ and $\boldsymbol{a} = (x, y) \in \mathbb{R}^2$ be bounded by the corner points $x_1, x_2, y_1, y_2 \in \mathbb{R}^2$ such that $x \in [x_1, x_2]$ and $y \in [y_1, y_2]$. The bilinear interpolant at point $\boldsymbol{x}$ is defined as

$$\begin{aligned} q_1(\boldsymbol{a}) = \frac{1}{\Delta x \Delta y}\Big(&(x_2 - x)\big((y_2 - y)f(x_1, y_1) + (y - y_1)f(x_1, y_2)\big) \\ &+ (x - x_1)\big((y_2 - y)f(x_2, y_1) + (y - y_1)f(x_2, y_2)\big)\Big), \end{aligned}$$

where $\Delta x = (x_2 - x_1)$, $\Delta y = (y_2 - y_1)$, and $q_1(\boldsymbol{a}) \approx g(\boldsymbol{a})$ [3, p. 159].

Lastly, *trilinear interpolation* linearly approximates a function value in $\mathbb{R}^3$. Let $h : \mathbb{R}^3 \to \mathbb{R}$ and $\boldsymbol{b} = (x, y, z) \in \mathbb{R}^3$ be bounded by the corner points $x_1, x_2, y_1, y_2, z_1, z_2 \in \mathbb{R}^3$ such that $x \in [x_1, x_2]$, $y \in [y_1, y_2]$ and $z \in [z_1, z_2]$. Now,

$$\begin{aligned} r_1(\boldsymbol{x}) = \frac{1}{\Delta x \Delta y \Delta z}\Big((x_2 - x) \quad &\Big((y_2 - y)\big((z_2 - z)f(x_1, y_1, z_1) + (z - z_1)f(x_1, y_1, z_2)\big) \\ &+(y - y_1)\big(z_2 - z)f(x_1, y_2, z_1) + (z - z_1)f(x_1, y_2, z_2)\big)\Big) \\ + (x - x_1) \quad &\Big((y_2 - y)\big((z_2 - z)f(x_2, y_1, z_1) + (z - z_1)f(x_2, y_1, z_2)\big) \\ &+(y - y_1)\big((z_2 - z)f(x_2, y_2, z_1) + (z - z_1)f(x_2, y_2, z_2)\big)\Big)\Big), \end{aligned}$$

where $\Delta x = (x_2 - x_1)$, $\Delta y = (y_2 - y_1)$, $\Delta z = (z_2 - z_1)$, and $r_1(\boldsymbol{b}) \approx h(\boldsymbol{b})$ is the linear approximation in $\mathbb{R}^3$ [9].

3.6.2 Weighted average

We use the definitions in [44] to formulate the weighted average. For a set of elements $A = \{a_1, \ldots, a_k\} \subseteq \mathbb{R}$, the *arithmetic weighted mean* is

$$\frac{\sum_{i=1}^{k} w_i a_i}{\sum_{i=1}^{k} w_i},$$

where $w_i \in \mathbb{R}$ is a weight corresponding to each a_i. Instead of an enumerated set of weights, we can have a continuous *weighting function* $\theta : \mathbb{R} \to \mathbb{R}$ which maps any $a \in \mathbb{R}$ to a corresponding weight. This yields another form for the arithmetic weighted mean, namely

$$\frac{\sum_{i=1}^{k} \theta(a_i) a_i}{\sum_{i=1}^{k} \theta(a_i)}.$$

Using the Euclidian norm defined in Equation (3.6), we define the *weighted mean distance* from a set of points $P = \{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_k\}$ to a given point $\boldsymbol{x} \in \mathbb{R}^n$ as

$$\frac{\sum_{i=1}^{k} \theta(\|\boldsymbol{p}_i - \boldsymbol{x}\|)\|\boldsymbol{p}_i - \boldsymbol{x}\|}{\sum_{i=1}^{k} \theta(\|\boldsymbol{p}_i - \boldsymbol{x}\|)}. \tag{3.21}$$

Furthermore, by substituting the latter instance of $\|\boldsymbol{p}_i - \boldsymbol{x}\|$ in the numerator in Equation (3.21), we obtain the distance weighted average of the point P locations at point $\boldsymbol{x}$, i.e.,

$$\frac{\sum_{i=1}^{k} \theta(\|\boldsymbol{p}_i - \boldsymbol{x}\|)\boldsymbol{p}_i}{\sum_{i=1}^{k} \theta(\|\boldsymbol{p}_i - \boldsymbol{x}\|)}. \tag{3.22}$$

3.6.3 Least squares methods

Least squares methods are methods of minimizing the squared distance between a model and data. Least squares can be seen as a problem of fitting a model to data. The minimization problem can be subject to some constraints. We consider the least squares problem with the Euclidian norm [8, pp. 204–205].

The basic form of a least square problem is the following: find the parameters of the function $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^n$ such that

$$\|\boldsymbol{f}(\boldsymbol{x}) - \boldsymbol{x}\|^2 = \sum_{i=1}^{n} (f_1(x_1) - x_1)^2 = (\boldsymbol{f}(x) - \boldsymbol{x})^T (\boldsymbol{f}(x) - \boldsymbol{x}),$$

is minimized. The minimization problem with the Euclidian norm is subject to finding the roots of the gradient with regards to the function f, namely the solutions of the the following equation:

$$\frac{d\|\boldsymbol{f}(\boldsymbol{x})-\boldsymbol{x})\|^2}{d\boldsymbol{f}} = \frac{(\boldsymbol{f}(\boldsymbol{x})-\boldsymbol{x})^T(\boldsymbol{f}(\boldsymbol{x})-\boldsymbol{x})}{d\boldsymbol{f}} = 0.$$

3.6.4 Newton's method

Let $f : \mathbb{R} \to \mathbb{R}$ be a C^n, $n \geq 1$ continuous function, thus $f'(x)$ exists for all $x \in \mathbb{R}$ [8, p. 58]. In order to find a root α for $f(\alpha) = 0$, an initial guess x_0 close to the root is taken. An iterative estimate

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \tag{3.23}$$

is the next estimate closer to the root α when some conditions are met. If there exists a neighborhood of α where $f(x)$, $f'(x)$, and $f''(x)$ are continuous, and $f'(\alpha) \neq 0$, then with an initial guess x_0 sufficiently close to α, the iterates in Equation (3.23) have a quadratic order of convergence, i.e., for every iterate there exists $c \in \mathbb{R}^+$ such that $\|\alpha - x_{n+1}\| \leq c\|\alpha - x_n\|^2$ [8, pp. 56, 60–61]. We omit the exact definition of sufficiently close and refer to [8] for further discussion on the initial guess x_0. Newton's method can be trivially extended to consider intersections between two functions $f(x)$ and $g(x)$ by setting $h(x) = f(x) - g(x)$ and solving α for $h(\alpha) = 0$.

We extend the Newton's method further to $\mathbb{R}^3$ by considering the intersection between a ray $\boldsymbol{r} : \mathbb{R} \to \mathbb{R}^3$ in parametric form $\boldsymbol{r}(t) = \boldsymbol{o} + \boldsymbol{d}t$ and a surface S. The surface is defined by a signed distance function $s(\boldsymbol{p})$ indicating the shortest projected distance of any point $\boldsymbol{p} \in \mathbb{R}^3$ to a tangent plane of the surface with unit normal $\boldsymbol{n}_T(\boldsymbol{x})$. These define a local surface plane near $\boldsymbol{p}$ with the implicit form $f_{\boldsymbol{p}}(\boldsymbol{x}) = \boldsymbol{n}_T \cdot (\boldsymbol{x} - \boldsymbol{p}) + s(\boldsymbol{p}) = 0$, where $\boldsymbol{x}$ is a point on the surface plane.

By solving $f_{\boldsymbol{p}}(\boldsymbol{r}(t)) = 0$ for t we get $t = \dfrac{\boldsymbol{n}_T(\boldsymbol{p}) \cdot (\boldsymbol{p} - \boldsymbol{o}) - s(\boldsymbol{p})}{\boldsymbol{n}_T(\boldsymbol{p}) \cdot \boldsymbol{d}}$, which gives us the value of t at the intersection between the ray $\boldsymbol{r}(t)$ and the tangent plane $f_{\boldsymbol{p}}(\boldsymbol{x}) = 0$. We formalize this procedure into the form of Equation (3.23): given an initial estimate t_0 along the ray $\boldsymbol{r}(t_0) = \boldsymbol{p}_0$ near an intersection $\boldsymbol{\alpha}$ at value $t_{\boldsymbol{\alpha}}$, the iterative estimate is given by

$$t_{n+1} = \frac{\boldsymbol{n}_T(\boldsymbol{r}(t_n)) \cdot \boldsymbol{d}t_n - s(\boldsymbol{r}(t_n))}{\boldsymbol{n}_T(\boldsymbol{r}(t_n)) \cdot \boldsymbol{d}}, \tag{3.24}$$

where $n \in \mathbb{R}$ and t_{n+1} is the next estimate closer to $t_{\boldsymbol{\alpha}}$.

4 PHYSICALLY BASED RENDERING

The basis of physically based rendering (PBR) is to model the behaviour of light in a virtual scene as accurately as possible. The scene can consist of a variety of geometry, materials and lights which makes modelling light interactions in the scene complex [62]. Lighting interactions in PBR are modelled by the behaviour of light on or in different materials at different angles. In real-time applications this comes down to *approximating* the behaviour as faithfully as possible. This means using numerical approximations, statistical and stochastic sampling, and error, variance and bias control to find the best approximate solutions in the real-time rendering budget [3, pp. 407–417].

4.1 Lighting interactions

The abstraction for light in PBR is a ray carrying lighting information of a set of light waves [4, p. 293–297]. It may be a light ray with certain wavelength if considering spectral rendering [49, pp. 261–263], but generally it is a light ray with direction and light intensity in some color space like in the red-green-blue (RGB) color space. Another abstraction needed is the point of interaction between the light ray and the participating surface or volume material. This material defines how a light ray interacts with the surface or volume, e.g., how the light ray reflects off a surface or refracts inside a volume.

Incoming and outgoing directions of a light ray have to be known to evaluate the light-surface interaction. The amount of refracted light is dependent on the angle of the light ray hitting the surface with transparent objects [49, pp. 442–446]. This is also true for opaque objects where the incoming angle or *incident angle* of the light ray affects the intensity of the reflective light component. The phenomenon is observable in the *Fresnel effect*: if light hits an opaque surface in a small incident angle (grazing angle) much of the light is almost perfectly reflected even on rough surfaces, which normally reflect light in diffuse angles with larger incident angles [49, pp. 460–462].

Interactions between surface materials and light rays are weighted by a distribution function called the bidirectional surface distribution function (BSDF). This distribution weights the contribution of a single light ray based on the incoming and outgoing direction of light. The distribution can be based on actual measurements of materials in the real world [49, pp. 462–466]. Certain attributes can also be used to artificially control properties of the material and the BSDF. These attribute abstractions differ between renderers, but usually abstractions such as specularity, diffuseness and glossiness are used [3, p. 105] [30, p. 353].

4.1.1 Bidirectionality

Bidirectionality is inherent in the BSDF. Bidirectionality means that the result of the rendering equation for a screen pixel is the same whether we inspect the path of a single light ray starting from a light source or a virtual pixel. The bidirectionality property gives an important tool for evaluating the rendering equation: we can follow the path of a light ray to the virtual screen pixel in the reversed order [30, pp. 387–389].

Following the light ray starting from the virtual camera affords faster convergence. This is due to knowing beforehand that the light ray path will contribute to the final color of the virtual screen pixel. If we follow the light ray from the other direction, i.e., starting from the light source, it is not guaranteed that the light ray will contribute to anything visible on the screen. This is due to the fact that the light ray has to be sampled at every interaction step and thus the ray direction might end up in a non-visible area.

The use of bidirectionality also needs some guarantees of rays hitting light sources to ensure proper convergence. This is achieved by using next event estimation (NEE): for every light ray-surface interaction we always sample the direction of a light source in addition to the randomly sampled direction [3, pp. 412–413] [30, pp. 851–854]. In a real-time PBR renderer this is done by sampling one extra direction for every surface interaction, where the sampled direction is the direction of a random light source. If the direction from the surface interaction to the random light source is unobstructed, then that direction contributes the intensity and color of the light source to the integral (weighted by the BSDF). Obstructed directions will also be beneficial because they can be handled as shadowed directions, and thus generate shadow effect to the surface interaction point. This is why the directions sampled by NEE are called shadow rays [30, p. 415].

4.2 Object materials

The photo-realism generated by PBR is based on the accurate modelling of surface and volume materials and their interaction with light [49]. Materials can be approximately described by basic attributes like metalness and roughness, which affect the scattering and absorption of light on the surface. However, to capture the accurate behaviour of light on materials, we need more complex distribution functions to describe the statistical behavior of light rays on the surface [30, p. 713].

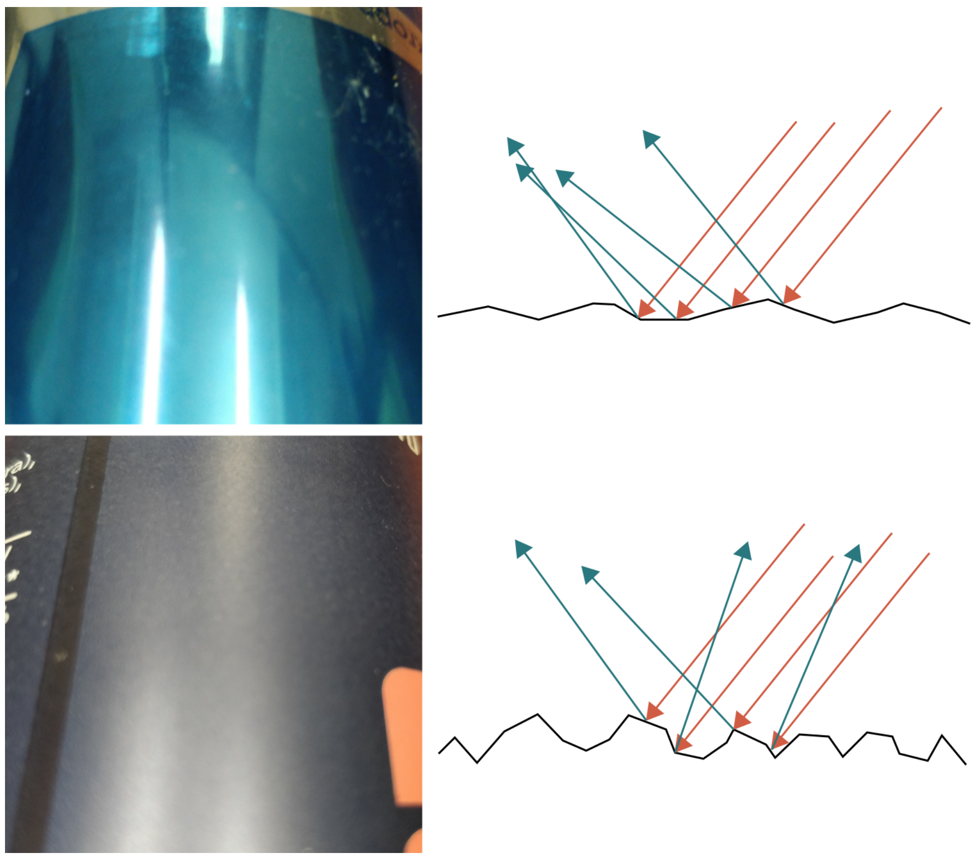

Figure 4.1. *The micro-level contours of a surface determine whether light rays (red arrows) reflect (blue arrows) to a similar direction (top-right) or scatter to more random directions (bottom-right). Two aluminium cans are photographed from the same position and angle (left): light scatters to the direction of a specular reflection on the smoother can surface and produces a sharper mirroring effect (top-left) compared to the rougher can surface that makes the reflections blurrier and the surface more matte (bottom-left). Adapted from [4, p. 305].*

4.2.1 Surface microstructure

Photorealistic material effects are produced by accurate surface representations. In real life, object surfaces have very intricate surface patterns that produce the visual appearance of the specific material. These patterns might be visible to the naked eye but generally they are only seen on a microscopic level [30, p. 711].

When a light ray hits a surface it can potentially interact with the surface several times. The light ray enters the microscopic contours of the surface and bounces around inside the surface before scattering back into the surroundings as illustrated in Figure 4.1. This interaction is called *micro-surface scattering*. Because modelling and rendering accurate micro-surface geometry is practically impossible, the microscopic structures have to be abstracted somehow. This is done with the bidirectional reflectance distribution function (BRDF) which models the distribution of light from a surface depending on the incoming and outgoing direction of the scattering light [30, p. 712].

4.2.2 Bi-directional surface distribution functions

The BSDF defines the interaction between a material surface and light. Light is described by a probability distribution where the probability varies based on the direction of both incoming and outgoing light. This is why the distribution is called *bidirectional*. The direction can be defined as a direction vector on a unit sphere.

Usually the BSDF is split into two parts: the scattering (BRDF) and transmittance part (bidirectional transmittance distribution function, BTDF) [30, p. 712]. The scattering part defines how light is distributed on the hemisphere oriented in the direction of the surface normal. Conversely the transmittance part defines the distribution inside the surface, namely on a hemisphere in the opposite direction of the surface normal. This division makes it easier to define materials, because for opaque materials, which don't let any light through, the BTDF part can be omitted. Furthermore the two parts have inherently different kinds of distributions and so it is easier to have a piece-wise definition of the whole distribution.

4.3 The rendering equation

In 1986 Kajiya introduced the *rendering equation* as the basis of physically correct interaction between light and surfaces in computer graphics [32]. The idea was to physically model the contribution of different direct and indirect light sources at individual points on a surface. This amounted to an integral over all possible directions in 3D-space from the surface point, namely a surface integral over a sphere's surface, which is formulated in the rendering equation as follows:

$$I_o(\boldsymbol{p},\boldsymbol{v}) = I_e(\boldsymbol{p},\boldsymbol{v}) + \int_{\boldsymbol{l}\in\Omega} f(\boldsymbol{l},\boldsymbol{v}) I_i(\boldsymbol{p},\boldsymbol{l})(\boldsymbol{n}_{\boldsymbol{p}}\cdot\boldsymbol{l})d\boldsymbol{l}, \tag{4.1}$$

where

$\boldsymbol{p}\in\mathbb{R}^3$	is the evaluation point on the surface
$\boldsymbol{v}\in\mathbb{R}^3$	is the unit direction vector of outgoing light
$\boldsymbol{n}_{\boldsymbol{p}}\in\mathbb{R}^3$	is the surface normal unit vector at evaluation point $\boldsymbol{p}$
$\boldsymbol{l}\in\mathbb{R}^3$	is the unit direction vector of incoming light
$I_o(\boldsymbol{p},\boldsymbol{v})$	is the outgoing light intensity from $\boldsymbol{p}$ to direction $\boldsymbol{v}$
$I_e(\boldsymbol{p},\boldsymbol{v})$	is the emitted light intensity from $\boldsymbol{p}$ to direction $\boldsymbol{v}$
$f:\mathbb{R}^3\times\mathbb{R}^3\to\mathbb{R}$	is a weighting function
$I_i(\boldsymbol{p},\boldsymbol{l})$	is the intensity of incoming light to $\boldsymbol{p}$ from direction $\boldsymbol{l}$
$\Omega\subset\mathbb{R}^3$	is the unit sphere centered at the origin

The integral part of Equation (4.1) is further split into two integrals over the hemisphere outside the surface (Ω^+) and the hemisphere inside the surface (Ω^-):

$$\int_{\boldsymbol{l}\in\Omega} f(\boldsymbol{l},\boldsymbol{v})I_i(\boldsymbol{p},\boldsymbol{l})(\boldsymbol{n_p}\cdot\boldsymbol{l})d\boldsymbol{l} = \int_{\boldsymbol{l}\in\Omega^+} f(\boldsymbol{l},\boldsymbol{v})I_i(\boldsymbol{p},\boldsymbol{l})(\boldsymbol{n_p}\cdot\boldsymbol{l})d\boldsymbol{l} + \int_{\boldsymbol{l}\in\Omega^-} f(\boldsymbol{l},\boldsymbol{v})I_i(\boldsymbol{p},\boldsymbol{l})(\boldsymbol{n_p}\cdot\boldsymbol{l})d\boldsymbol{l}.$$

With this formulation it is possible to consider the interaction of light inside and outside an object surface separately. Furthermore, because both $\boldsymbol{n_p}$ and $\boldsymbol{l}$ are unit vectors, by Equation (3.7) $\boldsymbol{n_p}\cdot\boldsymbol{l} = \cos\alpha$, where α is the angle between the two vectors.

4.3.1 Sampling the integral

Generally the evaluation of the rendering equation in analytical form is not possible. This can be illustrated by analyzing the components of the integral part of Equation (4.1). We concentrate on the most challenging component of the integral, which is I_i.

To evaluate the value of $I_i(\boldsymbol{p},\boldsymbol{l})$ in every direction $\boldsymbol{l}\in\Omega$ we would need to know where the light originates from in every direction. The incoming light intensity from direction $\boldsymbol{l}$ is equal to the outgoing light intensity from the closest surface point $\boldsymbol{p_l}$ intersected by the ray $\boldsymbol{p}+\boldsymbol{l}t$ to direction $-\boldsymbol{l}$, i.e., $I_i(\boldsymbol{p},\boldsymbol{l}) = I_o(\boldsymbol{p_l},-\boldsymbol{l})$ [4]. The outgoing light $I_o(\boldsymbol{p_l},-\boldsymbol{l})$ from $\boldsymbol{p_l}$ is another rendering equation, evaluated in the same manner. This equates to a recursive function

$$I_o(\boldsymbol{p},\boldsymbol{v}) = I_e(\boldsymbol{p},\boldsymbol{v}) + \int_{\boldsymbol{l}\in\Omega} f(\boldsymbol{l},\boldsymbol{v})I_o(\boldsymbol{p_l},-\boldsymbol{l})(\boldsymbol{n_p}\cdot\boldsymbol{l})d\boldsymbol{l}.$$

The recursion stops for one instance of I_o if at some evaluation point for all directions $\boldsymbol{l}\in\Omega$ there doesn't exist an intersection point $\boldsymbol{p_l}$. Then, only the emitting component I_e has to be evaluated. However, as the integral is over all direction vectors $\boldsymbol{l}$ on the unit sphere Ω, it is highly probable that some object surface is intersected and the recursion growth is unbound. The problems of evaluating the rendering equation have been noted in [30, p. 801, 825].

Numerical methods can be used to approximate the result of the integral part of the rendering equation [49, p. 12]. One solution category is importance sampling (Section (3.5.2)). Instead of taking uniformly distributed samples on the unit sphere, we can draw samples distributed by the BSDF $f(\boldsymbol{l},\boldsymbol{v})$. Applying Equation (3.19), we obtain an importance sampled approximation

$$I_o(\boldsymbol{p},\boldsymbol{v}) \approx I_e(\boldsymbol{p},\boldsymbol{v}) + \frac{1}{n}\sum_{i=1}^{n} I_o(\boldsymbol{p}_{\boldsymbol{l}_i},-\boldsymbol{l_i})(\boldsymbol{n_p}\cdot\boldsymbol{l}_i), \tag{4.2}$$

where $\boldsymbol{l}_i$ is distributed by the BSDF at $\boldsymbol{p}$ and outgoing direction $\boldsymbol{v}$.

4.4 Path tracing

GI effects are the hallmark of PBR and give it a photorealistic visual appearance. Unfortunately producing realistic GI is hard with traditional rasterization techniques. Also simple ray tracing techniques are not enough if only shadow rays and specular reflections are traced. Path tracing is the technique of recursively ray tracing every relevant direction of light at each surface-light intersection point.

Ray tracing techniques rely on finding intersections between rays and scene objects. These intersections can be computed with various methods which can be categorized in at least two different ways. The first categorization is concerned with whether the method results in an explicit location of intersection, e.g., by solving the roots of a polynomial, or whether it implicitly finds a close intersection by deciding if a ray has passed an object close enough to qualify as an intersection. The latter one can be implemented with *ray marching* [4, pp. 217–218] or *signed distance field* [4, p. 454] techniques.

Intersection testing, namely finding the closest ray-surface intersection, is usually the most time consuming part of ray tracing [49, p. 192] and for this purpose efficient acceleration structures have been developed [3, p. 416]. In general, acceleration structures try to encapsulate a complex virtual scene in a hierarchical representation which has simpler intersection computations with a ray which is illustrated in Figure 1.3. When the scene is divided into hierarchical parts, the intersection testing consists of checking intersection first with the parent node and then recursively with each child node within it. The recursion stops when it either reaches a leaf node or an empty node (a leaf node also, to be exact). Then the ray is intersected with a leaf node or declared a miss with an empty node. Acceleration structures decrease the complexity of brute force intersection testing up to logarithmic order $O(\log n)$ when the hierarchy structure is *balanced* [3, p. 415]. They also offer early exits for rays which enter empty areas in the virtual scene.

Even though acceleration structures reduce the asymptotic complexity of intersection testing, they don't provide reductions to the actual ray-primitive intersection algorithm. The intersection test implementation and complexity varies significantly based on the primitives being intersected [49, pp. 107–174].

When ray tracing point clouds, there exists no explicit surface which has to be intersected faithfully, so it is possible to use approximate intersections. Furthermore, the divisions made in the acceleration structure hierarchy don't have to take connectivity and topological information into account for points , so the construction of such hierarchies is less bound [26, pp. 5–6].

Rays are categorized into two groups: primary or camera rays and secondary rays. Primary rays are the ones cast from the virtual camera center through the virtual screen into the virtual world. The rays are defined by their origin (camera center) and their direction (virtual screen pixel and camera center difference). In the case of secondary rays the origin is the intersection point between an incoming ray and a model surface point it has

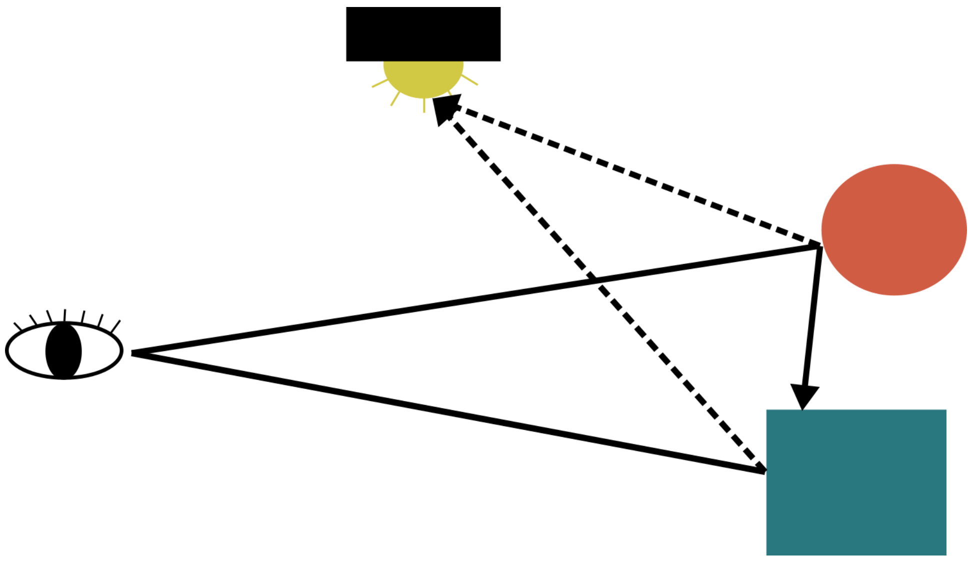

Figure 4.2. *Primary rays (solid line starting from eye) are cast from the virtual camera position through the pixels of a virtual monitor. Primary rays intersect with the objects in the scene and secondary rays are generated from the intersection points. Secondary rays include shadow rays (dashed lines) cast towards light sources, refraction rays cast inside the objects, and reflection rays (solid line from red ball) stochastically cast based on the object material BRDF at the intersection.*

hit, which is illustrated in Figure 4.2. The direction of the secondary ray is decided by two factors. Firstly if we want to prioritize direct illumination in favor of faster sample convergence, we cast a shadow ray towards a random light source. This process decides the direction of the secondary ray. In a scene with only one point light source this would mean that the direction of the shadow ray would always be fixed towards the point light.

The second factor in determining the direction of a secondary ray is by sampling the BSDF (Section 4.2.2) of the material of the intersection. The BSDF is a function which defines the probability distribution of the directions in which a light ray can be reflected or refracted from a given material. As discussed in Section 4.2.2, the BSDF is further divided into the BRDF (reflectance part) and the BTDF (transmitted/refracted part). The inputs of a BSDF are the point of intersection, incident angle of the incoming ray and the outgoing angle of the ray. The output is a probability value which is used to weigh the contribution of the particular sample. As can be seen from the rendering equation (Equation (4.1)), the result of the integral part is the sum of the BSDF weighted samples.

5 EFFICIENT POINT CLOUD RAY TRACING

A point cloud is defined as a set of points in 3D space, i.e., a set $P = \{\boldsymbol{p}_1, \dots, \boldsymbol{p}_n\} \subset \mathbb{R}^3$. The elements $\boldsymbol{p_i} \in \mathbb{R}^3$ of the point cloud explicitly define their positions in a Euclidian normed vector space, as defined in Section 3.3. For any element $\boldsymbol{p} := [p_x, p_y, p_z] \in P$, the components are the coefficients for each orthogonal basis vector in $\mathcal{E}^3$. These are the x, y and z coordinates of the point in the scene.

Point cloud points can also have attributes attached to them. The typical ones include color and surface normal or orientation of the point, but additionally there might be e.g. material, radius of influence and emittance information. Mathematically the attributes can be thought of as results of a mapping from the point cloud element to their attribute domain. For example, surface normals can be defined as a vector valued one-to-one mapping $\boldsymbol{n} : P \to \mathbb{R}^3$ which evaluates the surface normal for each element in P. In practice this means that the points are equipped with predefined attributes and the mapping is just a pointer or index to the attribute corresponding to the point.

In order to ray trace a point cloud, we need to define what constitutes an intersection between a point and a ray. A ray can be defined in parametric form (Section 3.4) as $\boldsymbol{r}(t) = \boldsymbol{o} + \boldsymbol{d}t$, where $\boldsymbol{o} \in \mathbb{R}^3$ is the origin of the ray, $\boldsymbol{d} \in \mathbb{R}^3$ is the direction of the ray and $t \in [t_{min}, t_{max}]$ is the parametric variable defining the beginning and end of the ray path. The interval $[t_{min}, t_{max}]$ can be set to $[0, \infty]$, but for efficiency they are set to correspond to the near and far plane distances.

In this Chapter the asymptotic analysis of the algorithms is based on the tools described in [66] and [18, pp. 43–64]. Specifically we use the Big O notation which is defined as follows:
Definition 5.1. Let $f : \mathbb{R} \to \mathbb{R}$ and $g : \mathbb{R} \to \mathbb{R}$. If there exists $c > 0$ and $n_0 \in \mathbb{N}$ such that for all $n > n_0$, $|f(n)| < cg(n)$, then we say that $f(n) = O(g(n))$.

We use the notation $T_i^j,\ i, j = 1, 2, \dots$ to refer to the execution time of line i in Algorithm j. The specific measurement of execution time is irrelevant because only the asymptotic behavior is analyzed. The asymptotic variables considered are the number of pixels M, point cloud points N, samples per pixel S, and recursive ray bounces B.

5.1 Surface reconstruction

Point clouds can describe an underlying surface or volume of a model with data points sampled from the surface or volume at certain positions. The sampling resolution, the average number of points in a given surface or volume area, is mostly dependent on the instruments and methods used to construct the point cloud. Generally point clouds can be generated in three different ways: sampling surfaces of real world objects with capturing devices like light detection and ranging (LIDAR) [45], reconstructing surface points from images or image sequences like multi-view stereo [11], and representing virtual design models or graphic models with densely sampled points [26, pp. 9–10]. Regardless of the point cloud origin the reconstruction benefits from being robust against ambiguous and undersampled point representations.

5.1.1 Mesh generation methods

The meshing methods fall into the category of explicit surface reconstruction. Explicit surface methods give surface representations which explicitly define the surface in question. Meshes are a sub-category which represents a surface with continuous piece-wise linear surface patches. Furthermore, the intermediate attributes within a surface patch are interpolated from the corner vertices (Section 3.6.1), which gives a smooth appearance to the mesh when rendered.

GPUs have been optimized to process triangle vertex primitives [4, pp. 29–55], which has contributed to the use of triangles as the typical mesh primitive. A popular triangulation algorithm type is the Delaunay triangulation [26, p. 103]. For point clouds with N elements in $\mathbb{R}^3$, Fortune [23] reports that the particular algorithmic implementations for 3D Delaunay triangulation, namely *Randomized incremental* and *Gift wrapping* triangulation methods, have an asymptotic worst case computational complexity of $O(N^2)$ and $O(N^3)$, respectively. Furthermore, if the point cloud is uniformly distributed inside a sphere, then the computational complexity is $O(N \log(N))$ and $O(N)$, respectively. In a later publication that introduced another 3D triangulation method called the *Crust algorithm*, the following was stated: “As has been frequently observed, the worst-case complexity for the three-dimensional Delaunay triangulation almost never arises in practice” [6].

Even though the asymptotic complexity analysis of the Delaunay triangulation methods don’t suggest a feasible algorithm for surface reconstruction, a state-of-the-art method for mesh reconstruction reports a maximum running time of 0.167 seconds for $6.14\;10^5$ points [46]. Combining the methods with massive point clouds with a number of samples in the order of $10^6 - 10^9$ makes them undesirable for real-time rendering of unprocessed raw point clouds. Additionally, generating a mesh on the whole point cloud can be wasteful if some areas of the point cloud are never rendered due to occlusion or view position [26, p.106].

5.1.2 Moving least squares

One well researched surface reconstruction method is the MLS method [26, pp. 95, 109–126] [11]. The method describes an implicit projection or field function which maps points in a local neighborhood to the surface implied by the point cloud. In this way it is not an explicit surface definition such as a mesh reconstruction. The benefits of the MLS technique are its robustness against noise in the point data [16]. Alexa et al. coined the term *point set surfaces* (PSS) to refer to MLS surfaces [5].

Kobbelt describes the basic MLS surface reconstruction algorithm in his survey [35]. Given a point set $P = \{\boldsymbol{p}_0, \boldsymbol{p}_1, ..., \boldsymbol{p}_N\}$, a C^0 continuous function $\boldsymbol{f} : R^n \to R^n$, where $\boldsymbol{f}(\boldsymbol{p}) = \boldsymbol{p}$ for every $\boldsymbol{p} \in P$ which are on the surface, is called the point set surface projection function of the surface S_P. In short this means that the function f defines the surface by including the points which project onto themselves. Finding the surface algorithmically involves a two step process, where we first find a linear reference domain for a point close to the assumed surface and then we approximate the local points once more with a higher order polynomial. The error minimization (least squares) in both approximation steps is controlled by the distance of local points (weighted appropriately with a positive monotonic function) to the reference domain and the local polynomial, respectively.

In the reference domain stage in order to find the domain $H = \{\boldsymbol{x} \,|\, \langle \boldsymbol{n}, \boldsymbol{x} \rangle - D = 0\}$, namely a plane with normal n and offset D, the following minimization problem is solved:

$$\arg\min_{\boldsymbol{n},D} \sum_i^N (\langle \boldsymbol{n}, \boldsymbol{p}_i \rangle - D)^2 \, \theta(\|\boldsymbol{p}_i - \boldsymbol{q}\|), \tag{5.1}$$

where $\boldsymbol{p}_i$ are the points in P, θ is the monotonic positive weighting function and $\boldsymbol{q} = \langle \boldsymbol{n}, \boldsymbol{r} \rangle - D$ is the projection of the evaluation point $\boldsymbol{r}$ on to the domain being approximated. Effectively this means minimizing the projected squared distance of each point in P weighted by the distance of the point from the projected point $\boldsymbol{q}$, which is thought of as the origin of the new domain H. In a practical implementation only an effective neighborhood of the evaluation point $\boldsymbol{r}$ is considered, because the weighting θ will be negligible for distant points and make them irrelevant in the minimization.

The local polynomial fitting phase is done in the reference domain defined in the previous phase. This means that the resulting polynomial approximation is not in the world coordinate domain, so the points in P have to be first projected into the reference domain. In this phase we solve the following minimization problem:

$$\arg\min_{g} \sum_i^N (g(x_i, y_i) - f_i)^2 \, \theta(\|\boldsymbol{p_i} - \boldsymbol{q}\|), \tag{5.2}$$

where x_i and y_i are the coordinates of the projection of $\boldsymbol{p_i}$ onto the reference domain H, and $f_i = \langle \boldsymbol{n}, \boldsymbol{p_i} - D \rangle$ is the distance of $\boldsymbol{p_i}$ to H. As a result we find the polynomial g defined in the local domain H, and we can project the local approximation of $\boldsymbol{r}$, namely

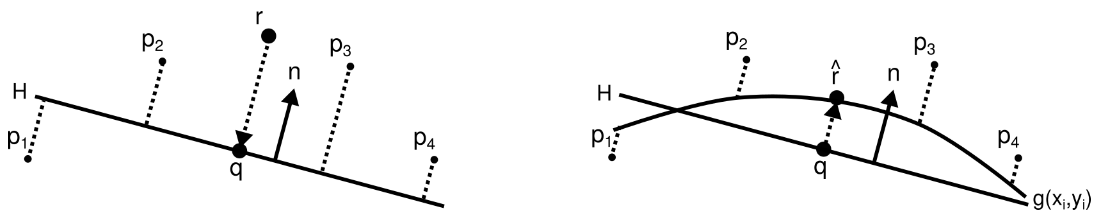

Figure 5.1. *The local MLS surface reconstruction is depicted. The surface is evaluated at point $\boldsymbol{r}$ by first fitting a local surface $H = \{\boldsymbol{x} \mid \langle \boldsymbol{n}, \boldsymbol{x} \rangle - D = 0\}$ by minimizing the weighted distances (dashed lines, left) of local points to the surface H. In the second stage, a polynomial $g(x_i, y_i)$ defined in the surface H coordinates is fitted to the local points by minimizing the weighted projected distance (dashed lines, right) of the points along $\boldsymbol{n}$ to g. Finally, the projected point $\hat{\boldsymbol{r}} = \boldsymbol{q} + g(0,0)\boldsymbol{n}$ on the local MLS surface is found (dashed arrow line, right).*

$\hat{\boldsymbol{r}}$ back to the world coordinate system by calculating $\hat{\boldsymbol{r}} = \boldsymbol{q} + g(0,0)\,\boldsymbol{n}$. The points $\boldsymbol{r} = \hat{\boldsymbol{r}}$ then define the local surface. The process is illustrated in 2D in Figure 5.1.

In a practical implementation for ray tracing, the projection from an evaluation point $\boldsymbol{r}(t)$ along a ray to the local domain H can be set to the nearest point $\boldsymbol{p}_i \in P$ as shown in [2]. Thus, the distance $\|\hat{\boldsymbol{r}}\|$ from a certain evaluation point $\boldsymbol{r}$ near a point $\boldsymbol{p}_i$ can be calculated and deemed to be on the MLS surface if the distance is close enough to zero.

5.2 Intersection evaluation

Intersection evaluation for point clouds is impossible without considering how to represent points or rays. Points have no volume or area in continuous three-dimensional space so the probability of intersecting them with a ray with no area or volume, namely a line, is zero. This is why either points or rays have to be expanded in a way that gives them an intersectable area or volume. This has been noted in some point cloud ray tracing publications [24, 34, 63].

Ray tracing techniques that use a radius for the ray primitive are quite common, which is discussed in Section 5.3. Sphere, cylinder, cone, and cube tracing are some prominent techniques for finding objects that are at a certain threshold distance from the ray path. If the threshold is set to small enough, the registration of a close object can be interpreted as an intersection. The first point cloud ray tracing method used cylinder tracing to find dense enough areas in the point cloud to be considered as intersections of the underlying surface [55]. A cone tracing method was proposed by Wand et al. [64] but it was an offline ray tracing system.

Making points "bigger" by generating a local linear surface around them is known as splatting. The basic idea is to define a plane with the point on the plane and the normal direction as either the point’s normal attribute or as the local curvature around the point neighborhood. The latter normal definition is produced by gradient calculations between the neighboring point cloud points. Additionally the plane has to be bounded to some locality of the point of origin.

5.2.1 Splatting

A common method of rendering point clouds is *splatting* [26, p. 247]. Splatting is based on approximating a point's local surface with a bound surface element. These surface elements are typically circular or elliptical oriented discs. The disc parameters, such as principal axis directions and lengths, are defined based on local surface curvature. Because no explicit surface exists in a point cloud representation, the curvature has to be approximated by some a set of points.

Splatting was applied in 2001 by Zwicker et al. [68] and it was used to raster point clouds with millions of individual points. The publication suggested changing the standard rendering primitive of triangles to point-based primitives. The benefit was purportedly the C^{-1} continuity[1] of point primitives, which gives freedom compared to the C^0 continuity[2] of triangle or other polygon meshes. The C^{-1} continuity of the splatting method also poses problems, one of which are hole artifacts present between splats [14]. When rasterizing, this is tackled with elliptical weighted average (EWA) filtering which is used for both splat and texture filtering [68].

In order to be faithful to sharp features, including edges and corners, splat-based rendering has to take into account discontinuities in the underlying surface. Naive implementation without edge and corner handling will lead to artifacts in edge areas where splats will incorrectly cross over edges unable to replicate sharp features. Surface splat clipping methods have been developed to deal with such artifacts. We refer to [31] for implementation details.

The benefit of splatting is seen especially in rasterization pipelines compared to ray tracing [26, p. 248]. Elliptical discs are easy to project from the world space to the screen space and with appropriate EWA filtering the end result is qualitatively good. The elliptical representation transfers to ray tracing pipelines as well, because elliptical surfaces are easy to intersect and check for boundaries. There is also the additional benefit of not needing to construct an explicit surface beforehand, which would then be intersected by the rays. In surface reconstruction, some areas of the point cloud model might be wastefully constructed even though those surfaces might never be intersected due to occlusions. Splatting gives the benefit of only constructing a surface representation when needed.

[1] C^{-1} continuous surfaces are discontinuous at least at one point

[2] C^0 surfaces are continuous but not differentiable at all points on the surface

5.3 Surveyed methods

A ray tracing method for direct point cloud rendering without splat or volume primitives was introduced by Schaufler & Jensen [55]. They used a derivation of ray tracing called cylinder tracing and used it to register intersections with the point cloud model when point density was high enough in the ray tube. Point attributes were weighted based on their projected distance from the ray cylinder center. Furthermore an octree acceleration structure was used to traverse the point cloud efficiently and find points near the ray radius.

Adamson and Alexa published a paper few years later introducing a method using moving least squares (MLS) surface reconstruction for point set surface ray tracing [2]. The point set surface definition was already introduced by Alexa et al. in 2001 [5] and it differed from a point cloud definition by the assumption that a set of points was also accompanied with a surface approximation near the data points for example by an MLS surface. The MLS surface was computed at rendering time.

A hybrid method combining both point and triangle primitives introduced a way of using a multi-resolution approach to ray tracing. A lower resolution representation approximated triangles with points and triangles were used for high resolution detail. The specific ray tracing method used was cone tracing accompanied with an octree acceleration structure for faster ray traversal [64].

To address time varying, animated and deforming point clouds Adams et al. suggested a surfel primitive based method in the context of ray tracing [1]. The surfels were bounded in a sphere hierarchy for acceleration and were interpolated iteratively based on neighboring surfels close to the surface-ray-intersection.

Wald and Seidel introduced the first real time ray tracing method for point clouds [63]. The method used simple Phong shading, hard shadow effects and splat culling by traversing implicit surfaces and splat primitives in a Kd-tree acceleration structure. It was able to render up to 20 frames per second (fps) of a point cloud with a million data points on consumer hardware in 2005.

Methods for ray tracing compressed point clouds were discussed in two publications by Hubo et al. [28, 29]. The former publication used a Kd-tree and a LOD system to ray trace splat primitives generated from the point cloud. The latter publication discussed a compression method for point clouds with *random access* ability to surface sections with applications for ray-surface intersections.

Possibly the first GPU-based ray tracer for point clouds achieving real-time frame rates was published by Tejada et al. [58]. It achieved up to 6.6 fps with reflection rays with limited depth. The point cloud's intersectable surface was produced by a MLS estimate, which approximated a point's local neighborhood with a bivariate second order polynomial. Primary intersection locations were refined with a two or three step iterative method, but secondary rays, including reflection and shadow rays, were cast from the non-refined rough intersection locations. The method lacked an acceleration structure for secondary ray traversal, so the authors followed with an improved method with a grid-based acceleration structure and added support for refracted rays [59]. The authors reported a performance of 10 fps with shadow rays and non-real-time frame rates with refracted rays.

In the beginning of the 2010s Goradia et al. and Kashyap et al. published several point cloud ray tracing methods achieving real-time frame rates [24, 25, 33, 34]. Previous point cloud ray tracing methods were implemented on the CPU and only one of them achieved real time frame rates [63].

Kashyap et al. initial paper achieved real-time frame rates of up to 55 fps by introducing the first GPU-based method for point cloud ray tracing using a textured splat octree and ray tracing for shadow rays [34]. Each pixel was sampled at 4 samples per pixel (spp) and 10 bounces for global illumination effects and the octree was constructed as a preprocessing step.

Goradia et al. followed with a paper that achieved real-time frame rates of up to 13 fps by introducing a GPU-based method for point cloud ray tracing using Phong shading for primary rays and ray tracing for shadow rays [24]. A high resolution implementation with more than one spp wasn't able to achieve real-time rendering speed.

Kashyap et al. published a method for ray tracing implicit surfaces produced from point samples. The method was able to produce advanced lighting effects such as reflections and refractions at real time frame rates of up to 46 fps by using an octree accelerated implicit surface with intersection interpolation. Generating the octree for the implicit surface representation had to be done before the rendering as a preprocessing step which took tens of seconds [33].

In the publication by Goradia et al., further advanced lighting effects including specular reflections and caustics were implemented in a real-time point cloud ray tracer achieving up to 20 fps with $5 \cdot 10^6$ points and 512×512 resolution [25]. The method used implicit surface octrees similar to [33] to accelerate intersection testing and precalculated caustic effects into a caustic sample map. Similar to previous publications, the method didn't achieve real-time frame rates with higher resolutions and more than one spp.

Splats were used as a point primitive in the publication by Cai et al. [14] which achieved real-time frame rates of up to 2 fps. A Kd-tree was used to accelerate an iterative procedure of ray-splat intersections which converged to an average surface normal based on neighboring splats. Another publication by Cai et al. described a nearest neighbor approach to point cloud ray tracing [15]. The basic method used sphere tracing to locate points near a traversing ray. The points were stored in a bounding sphere hierarchy to accelerate the traversal and additionally, an improved method generated triangle primitives between the points when intersected. Neither the basic or the improved method achieved real-time frame rates.

Categorization of the methods

Based on the survey, seven major method categories were identified. The most important category for the subject of this thesis is the real-time capability of the methods. In this thesis, a real-time rendering method is defined as a method achieving at least 1 fps, i.e., 1 second per frame on consumer hardware. The other categories are intersection acceleration structure utilization, point cloud preprocessing utilization, MLS surface definition, implicit surface definition, direct splat intersection, and iterative intersection refinement (see Section 3.6.4). Table 5.1 summarizes the major categorization of the methods.

Table 5.1. *A categorization table of the surveyed methods. The surveyed methods are categorized based on their real-time rendering capabilities, the use of intersection acceleration structures, and the utilization of additional point cloud preprocessing. Furthermore, the surveyed methods have three major surface definition categories including the MLS surface, implicit surface, and splat surface, which are intersected with iterative or non-iterative means.*

Method	Real-time	Acceleration	Preprocess	MLS surface	Implicit surface	Splat intersection	Iterative refinement
Schaufler and Jensen 2000 [55]		X					
Adamson and Alexa 2003 [2]		X	X	X			X
Wand and Straßer 2003 [64]		X	X				
Adams et al. 2005 [1]		X	X		X		X
Wald and Seidel 2005 [63]	X	X	X		X		X
Tejada et al. 2006 [58]	X		X	X			X
Hubo et al. 2006 [28]		X			X		X
Hubo et al. 2007 [29]	X	X	X		X		
Tejada et al. 2007 [59]	X	X	X	X			
Linsen et al. 2007 [41]		X	X			X	
Kashyap et al. 2010 [34]	X	X	X			X	
Goradia et al. 2010 [24]	X	X	X			X	
Kashyap et al. 2010 [33]	X	X	X		X		
Goradia et al. 2011 [25]	X	X	X		X		
Cai et al. 2011 [14]	X	X	X				X
Cai et al. 2015 [15]		X	X				X

Real-time methods include [14, 24, 25, 33, 34, 58, 59, 63] and non-real-time methods include [1, 2, 15, 28, 29, 41, 55, 64]. The observed trend is that the more recent publications from the mid 2000s to 2010s present real-time methods due to GPU implementations which massively distribute the work load of the ray tracing algorithm to parallel computational units. On the contrary, the publications from the early 2000s use serial implementations of the ray tracing algorithm on the central processing unit (CPU) and are thus not real-time, except for [63].

Intersection acceleration structures were used by all methods except Tejada et al. [58]. Furthermore, the improved method by Tejada et al. [59] used an acceleration structure only for secondary rays. Both of the methods used rasterization to find the closest intersection of the primary rays, which doesn't utilize intersection acceleration structures designed for ray tracing. See Section 5.4 for details on the intersection acceleration structures.

There are three major surface definitions for the point cloud model used by the methods. An MLS surface definition is utilized in [2, 58, 59], an implicit surface definition is used in [1, 25, 29, 33, 63], and a splat-based definition and intersection is used in [24, 34, 41, 64]. Three of the methods use a different approach to the surface definition, namely a point to ray coordinate projection [55], a height-difference variant of the implicit surface [14], and a k nearest neighbor (k-NN) surface and normal construction [15].

The intersection testing is iteratively refined in [1, 2, 14, 15, 58, 63]. Hubo et al. don't disclose how they find the exact intersection to their weighted implicit surface [29]. The improved method by Tejada et al. uses the first iterative MLS surface approximation as the final intersection [59]. The methods using a splat-based definition of the points intersect the splats directly without iterative refinement [24, 34, 41]. The rest of the methods either regularly sample a trilinearly interpolated implicit function [25, 33] or use a distance weighted average of point cloud points projected onto ray coordinates [55, 64].

The computational performance of the methods are summarized in Table 5.2. Furthermore, a summary of the prerequisites needed for rendering by the methods is shown in Table 5.3.

Table 5.2. *Rendering times per frame reported by the surveyed methods are shown in both best case and worst case configurations for rendering time. Note that the execution times are measured on varying consumer hardware at the time of the publications and cannot be directly compared.*

Method	**Best rendering time**				**Worst rendering time**			
	Time per frame (seconds)	**Resolution (pixels)**	**Point cloud size (points)**	**spp**	**Time per frame (seconds)**	**Resolution (pixels)**	**Point cloud size (points)**	**spp**
Schaufler and Jensen 2000 [55]	11 minutes	768×512	$3.5 \cdot 10^5$	4	61 minutes	1024×1024	$2 \cdot 10^6$	4
Adamson and Alexa 2003 [2]	-	-	-	-	hours	N/A	$1.0 \cdot 10^5$	N/A
Wand and Straßer 2003 [64]	99	512×512	$7.4 \cdot 10^3$	$1-?$ (adaptive)	1284	640×480	$8.7 \cdot 10^6$ triangles, $1.1 \cdot 10^5$	$1-?$ (adaptive)
Adams et al. 2005 [1]	6	512×384	$1.0 \cdot 10^5$ surfels	N/A	20	512×512	$2.7 \cdot 10^5$ surfels	N/A
Wald and Seidel 2005 [63]	0.18	512×512	$1.3 \cdot 10^6$	N/A	0.5	640×480	$2.4 \cdot 10^7$	N/A
Tejada et al. 2006 [58]	0.15	512×512	$3.6 \cdot 10^4$	N/A	0.870	512×512	$1.1 \cdot 10^5$	N/A
Hubo et al. 2006 [28]	9	512×512	$2.3 \cdot 10^8$	N/A	10	512×512	$3.7 \cdot 10^8$	N/A
Hubo et al. 2007 [29]	-	-	-	-	2	N/A	$2.8 \cdot 10^7$, $2.8 \cdot 10^5$ patches	N/A
Tejada et al. 2007 [59]	0.10	640×480	$3.6 \cdot 10^4$	N/A	4.5	640×480	$4.8 \cdot 10^4$	N/A
Linsen et al. 2007 [41]	1	150×150	$4.4 \cdot 10^5$, $2.9 \cdot 10^5$ splats	N/A	408	1200×1200	$6.6 \cdot 10^5$, $3.8 \cdot 10^5$ splats	N/A
Kashyap et al. 2010 [34]	0.018	512×512	10^6	N/A	0.25	512×512	$3.8 \cdot 10^6$	10
Goradia et al. 2010 [24]	0.072	512×512	$1.0 \cdot 10^6$	1	2.0	512×512	$1.3 \cdot 10^6$	16
Kashyap et al. 2010 [33]	0.050	512×512	$5 \cdot 10^6$, $2.9 \cdot 10^6$ reduced points	1	0.13	512×512	$1.5 \cdot 10^7$, $5.7 \cdot 10^6$ reduced points	4
Goradia et al. 2011 [25]	0.050	512×512	$5 \cdot 10^6$, $2.9 \cdot 10^6$ reduced points	1	0.13	512×512	$1.5 \cdot 10^7$, $5.7 \cdot 10^6$ reduced points	4
Cai et al. 2011 [14]	0.44	500×400	$8.1 \cdot 10^4$ splats	N/A	0.72	500×400	$2.0 \cdot 10^5$ splats	N/A
Cai et al. 2015 [15]	7.8	400×500	$3.4 \cdot 10^5$	N/A	100	500×400	$3.0 \cdot 10^6$	N/A

5.4 Accelerating intersection testing

When a ray passes through a virtual scene, it usually misses most of the geometry without intersecting them. To avoid iterating through all of the points in a point cloud and

Table 5.3. *Preconditions for ray tracing for the surveyed methods, not including intersection acceleration structures (see Table 5.4).*

Method	**Point radii**	**Point normals**	**Normal gradients**	**k-NN structure**	**Precomputed MLS surface**	**Precomputed implicit surface**
Schaufler and Jensen 2000 [55]		X				
Adamson and Alexa 2003 [2]						
Wand and Straßer 2003 [64]	X	X	X			
Adams et al. 2005 [1]	X	X				
Wald and Seidel 2005 [63]	X	X				
Tejada et al. 2006 [58]	X			X	X	
Hubo et al. 2006 [28]		X				
Hubo et al. 2007 [29]	X	X				
Tejada et al. 2007 [59]	X			X	X	
Linsen et al. 2007 [41]	X	X	X			
Kashyap et al. 2010 [34]	X	X				
Goradia et al. 2010 [24]	X	X				
Kashyap et al. 2010 [33]	X	X				X
Goradia et al. 2011 [25]	X	X				X
Cai et al. 2011 [14]	X	X				
Cai et al. 2015 [15]				X		

querying for an intersection, yielding $O(N)$ operations, acceleration structures are used. In general, acceleration structures partition the virtual scene into hierarchies or grids that bound the scene geometry into easily intersectable objects, like planes or spheres. The acceleration structures utilized in the surveyed methods are the *octree*, *kd-tree*, *bounding sphere hierarchy*, and grid structures, which are briefly introduced. The asymptotic computational complexity of constructing and querying intersections from the structures are presented.

All structures share in common a highest level bounding volume, which encapsulates all the geometry in the scene. This is a cube or a box in the case of octrees, kd-trees and grid structures. For bounding sphere hierarchies the whole hierarchy is inside a bounding sphere.

The kd-tree structure uses axis aligned planes to partition the geometry hierarchically. It uses a splitting plane to partition the geometry to two sections containing roughly the same amount of primitives. The two partitions are split again iteratively and this process is continued until some predefined condition, such as the number of primitives in a partition or depth of the hierarchy. The construction time of a kd-tree has been shown to have a theoretical lower bound $O(N \log N)$, which has been also achieved in practice [61].

In an octree structure, the topmost bounding cube (root) is split into eight equally sized smaller cubes (inner nodes), which are further split recursively in the same manner (Figure 1.3. Similar to the kd-tree, the recursion is stopped when a threshold of the number of primitives in a cell or recursion depth in an octree node is met. The octree can be constructed in $O(N \log N)$ time [7].

A bounding sphere hierarchy is a special case of a more general *bounding volume hierarchy* (BVH), where primitives are encapsulated in volume objects, like boxes or spheres, with dynamically varying bounds, contrary to the static structure of the octree. The volume objects are encapsulated from the highest level bounding sphere to the lowest with a threshold on the number of primitives in the lowest level sphere. Maximum number of levels can be set to limit the height of the hierarchy. The bounding volume hierarchy construction time in general is $O(N \log(N))$ as stated in [38].

A grid structure places a predefined number G of equally sized grid cells within the highest level bounding volume. Points within the highest level bounding volume are iterated over and placed into the correct grid, which yields an $O(N)$ construction time [22, p. 287]. For a uniformly distributed point cloud, a $K = N/G$ number of primitives are in each grid cell on average. By choosing G as a fraction of N, say $G = \frac{1}{c}N,\ c \geq 1$, we can achieve an average number of primitives in each grid cell irrespective of N.

Kd-trees, octrees and bounding sphere hierarchies all have a hierarchical structure, which enables nearest intersection queries down to the leaf node level in $O(\log N)$ time complexity on average [3, p. 647]. For grid acceleration, the average querying time complexity is constant, i.e., O(1) [57]. The surveyed methods using acceleration structures construct them before ray tracing as a preprocessing step.

Acceleration structures used by the surveyed methods are summarized in Table 5.4. Methods [58] and [59] don't use any acceleration structure because they find the intersection between rays and points by rasterizing the points to virtual pixels. This is discussed further in the algorithm complexity analysis in Section 5.5.

Other preprocessing algorithms

Most of the surveyed methods have to perform additional preprocessing to the point data before ray tracing. Splats are assumed or generated from raw point cloud data prior to ray tracing by methods [14, 24, 33, 34, 41, 63, 64]. Many of the methods use an optimal splat generation algorithm described in [67] to produce splats from point clouds.

A k-nearest neighbor (k-NN) structure is constructed by the methods [1, 15, 58, 59] for querying neighboring points or splats when evaluating MLS surfaces or weighted averages . Cai et al. [15] construct a *binary tree* with *photon mapping* , whereas the remaining methods don't describe their particular algorithm for the construction.

Other various preprocessing algorithms include patch based compression [29], normal field evaluation [41, 64], CSM octree [25], and splat octree refinement [24, 34].

The asymptotic analysis of the other preprocessing methods is out of scope in this thesis. The preprocessing algorithms for the different methods are summarized in Table 5.4.

5.5 Comparing the surveyed methods

The surveyed methods exhibit similarities specifically in their high level ray tracing rendering algorithm. In Section 5.5.1, we describe a generic ray tracing algorithm which encapsulates the high level functionality for all methods, except [58, 59]. Both meth-

Table 5.4. *The table shows the acceleration structures used for intersection testing and the computational complexity of constructing them. Other preprocessing algorithms utilized by the surveyed are also listed.*

Method	**Intersection acceleration**	**Construction complexity**	**Other preprocessing**
Schaufler and Jensen 2000 [55]	Octree	$O(N \log N)$	-
Adamson and Alexa 2003 [2]	Bounding sphere hierarchy	$O(N \log N)$	-
Wand and Straßer 2003 [64]	Octree	$O(N \log N)$	Splat generation, normal field
Adams et al. 2005 [1]	Bounding sphere hierarchy	$O(N \log N)$	k-NN
Wald and Seidel 2005 [63]	Kd-tree	$O(N \log N)$	Splat generation
Tejada et al. 2006 [58]	-	$O(1)$	k-NN
Hubo et al. 2006 [28]	Kd-tree	$O(N \log N)$	-
Hubo et al. 2007 [29]	Octree	$O(N \log N)$	patch compression
Tejada et al. 2007 [59]	Grid (for secondary rays)	$O(N)$	k-NN
Linsen et al. 2007 [41]	Octree	$O(N \log N)$	Splat generation, normal field
Kashyap et al. 2010 [34]	Octree	$O(N \log N)$	Splat generation, octree refinement
Goradia et al. 2010 [24]	Octree	$O(N \log N)$	Splat generation, octree refinement
Kashyap et al. 2010 [33]	Octree	$O(N \log N)$	Splat generation
Goradia et al. 2011 [25]	Octree	$O(N \log N)$	CSM octree
Cai et al. 2011 [14]	Kd-tree	$O(N \log N)$	Splat generation
Cai et al. 2015 [15]	Grid	$O(N)$	k-NN

ods by Tejada et al. use rasterization to simulate the intersections of rays and the virtual scene. A generic rasterization algorithm has average complexity of $O(MN)$ [39].

5.5.1 Generic ray tracing algorithm

Algorithm 1 shows in pseudocode the general ray tracing algorithm already discussed in Chapter 4. We identify the parts of the algorithm which have a varying execution time depending on the surveyed method.

In Algorithm 1, line 1 is executed S times and lines 2 to 5 are executed SM times in all cases so

$$T^1_{tot} = O(ST^1_1) + O(SMT^1_2) + O(SMT^1_3) + O(SMT^1_4) + O(SMT^1_5). \tag{5.3}$$

The execution time of T^1_1 and T^1_2 are independent of all the asymptotic variables, thus

```
Data: Virtual screen pixel coordinates $pixels$, camera origin $\boldsymbol{o}$, number of samples
      per pixel $S$, number of bounces $B$, and scene objects and lights $objects$
Result: Light intensity $I_o$ for each pixel
1 for $i \leftarrow 1$ to $S$ do
2     for $\boldsymbol{p} \in pixels$ do
3         $\boldsymbol{r} \leftarrow castRay(\boldsymbol{o}, \boldsymbol{p})$;
4         $\boldsymbol{x}, \boldsymbol{n} \leftarrow closestIntersection(\boldsymbol{r}, objects, t_{min}, t_{max})$;
5         $I_o[\boldsymbol{p}, i] \leftarrow evaluateRenderingEq(\boldsymbol{x}, \boldsymbol{n}, \boldsymbol{r}, B, objects, t_{min}, t_{max})$;
6     end
7 end
8 $I_o[\boldsymbol{p}] \leftarrow \frac{1}{S} \sum_{i=1}^{S} I_o[\boldsymbol{p}, i]$;
```

Algorithm 1: A high level algorithm of a general ray tracing algorithm.

$T_1^1 = O(1) = T_2^1$. Additionally the function $castRay$ on line 3 calculates the direction of the ray $\boldsymbol{r}$ by $\boldsymbol{d_r} = \dfrac{\boldsymbol{p}-\boldsymbol{o}}{\|\boldsymbol{p}-\boldsymbol{o}\|}$ and the origin $\boldsymbol{o_r} = \boldsymbol{o}$, which is independent of the asymptotic variables, i.e., $T_3^1 = O(1)$. Feeding this back to Equation (5.3) we have

$$T_{tot}^1 = O(S) + O(SM) + O(SM) + O(SMT_4^1) + O(SMT_5^1) = O(SMT_4^1) + O(SMT_5^1).$$

In Algorithm 1, the function $closestIntersection$ (line 4) finds the closest intersection between a ray and the scene. This means iterating through all primitives in the scene and checking if the ray intersected, which yields $O(N)$ complexity. If an intersection acceleration structure is present (see Section 5.4), then the closest intersection down to a leaf node can be queried in down to $O(\log N)$ time. We denote the query time of the closest intersection down to a leaf node by T_{query} and the final intersection time within the leaf node by T_{int}.

If a ray misses all points or primitives inside a leaf node, the intersection query has to be done again. This yields a worst case scenario where queries are done in the order of $O(N)$ times and the asymptotic complexity in total for $closestIntersection$ is $T_4^1 = O(N(T_{query} + T_{int})$, when an acceleration structure is used. However, this is rare and requires a low quality acceleration structure and a special configuration of scene objects . Furthermore, the worst case would dominate the asymptotic behavior of the ray tracing and make the analysis meaningless. Thus, we assume the average case of $O(T_{query})$ and deduce $T_4^1 = T_{query} + T_{int}$.

Recursive rendering equation evaluation

The function $evaluateRenderingEq$ in Algorithm 1 (line 5) evaluates the rendering equation (Equation (4.1)). As discussed in Chapter 4, the rendering equation has a recursive integral function which is approximated by generating one random sample direction at each level of recursion. Furthermore, by importance sampling the random directions from the BRDF, we sample the more contributing directions of the integral. The depth of the recursion of the integral is defined by the number of bounces B. This amounts to casting one ray from a previous level intersection point, finding the closest intersection, and evaluating the rendering equation at the new intersection point, B times recursively.

The pseudocode for the recursive function $\mathit{evaluateRenderingEq}$ is shown in Algorithm 2. It should be noted, that the algorithm doesn't exhibit shadow, reflective, or refractive rays. Furthermore, there is no modulation of the texture color of each intersection point. These details are omitted in both Algorithms 1 and 2 for clarity, because they don't affect the asymptotics of either algorithm.

```
1  Function evaluateRenderingEq($\boldsymbol{x}$, $\boldsymbol{n}$, $\boldsymbol{r}$, $B$, $objects$, $t_{min}$, $t_{max}$):
2      if $B \leq 0$ then
3          return $0$;
4      else if $\boldsymbol{x} = null$ then
5          return $0$;
6      else if $isLightSource(\boldsymbol{x})$ then
7          return $emittedLight(\boldsymbol{x}, \boldsymbol{r})$;
8      else
9          $E_o \leftarrow emittedLight(\boldsymbol{x}, \boldsymbol{r})$;
10         $f_{\boldsymbol{r}} \leftarrow BRDF(\boldsymbol{x}, \boldsymbol{r})$;
11         $\boldsymbol{l} \leftarrow distributedRand(f_{\boldsymbol{r}})$;
12         $\boldsymbol{r}' \leftarrow castRay(\boldsymbol{x}, \boldsymbol{l})$;
13         $\boldsymbol{x}', \boldsymbol{n}' \leftarrow closestIntersection(\boldsymbol{r}', objects, t_{min}, t_{max})$;
14         $I_o \leftarrow evaluateRenderingEq(\boldsymbol{x}', \boldsymbol{n}', \boldsymbol{r}', B - 1, objects, t_{min}, t_{max}) \cdot dot(\boldsymbol{n}, \boldsymbol{l})$;
15         return $E_o + I_o$;
16     end
```

Algorithm 2: Recursive function for evaluating the rendering equation in Equation (4.1)

In Algorithm 2, lines 2, 4, and 6 are conditions for terminating the recursion in the case of exceeding the number of bounces B, not finding an intersection at the previous level, or hitting a light source (function $\mathit{isLightSource}$), respectively. In the first two cases, 0 intensity is returned (lines 3 and 5). In the case of hitting a light source, only the emitting component of the light is returned (line 7), as any indirect light intensity at the light source is negligible compared to the emitted intensity.

The function $\mathit{emittedLight}$ in Algorithm 2 (line 9) evaluates the intensity of the emitted light at point $\boldsymbol{x}$ in the direction of the incoming ray $\boldsymbol{r}$ and $BRDF$ (line 10) fetches the distribution function at surface point $\boldsymbol{x}$ with incoming direction from ray $\boldsymbol{r}$. The function $\mathit{distributedRand}$ (line 11) produces a random unit direction vector drawn from the BRDF in $f_{\boldsymbol{r}}$ (importance sampling), and $\mathit{castRay}$ (line 12) is the same function as in Algorithm 1. These functions use the values stored in $\boldsymbol{x}$, thus, don't depend on the asymptotic variables, yielding constant time complexity $O(1)$.

The function $\mathit{closestIntersection}$ (line 11) is identical to the one presented in Algorithm 1 and has the same asymptotic complexity $T_{11}^2 = T_4^1 = T_{query} + T_{int}$. The recursive function call $\mathit{evaluateRenderingEq}$ (line 12) has the parameter $B - 1$ decremented from the previous recursion level. In total, the function will be called recursively in the order of $O(B)$ times until the condition $B \leq 0$ (line 2) is met in the worst case scenario. This yields a total asymptotic complexity of $T_5^1 = T_{tot}^2 = O(B(T_{query} + T_{int}))$ for Algorithm 2.

Combining the asymptotic analysis of Algorithms 1 and 2 gives us

$$T_{tot}^1 = O(SM(T_{query} + T_{int})) + O(SMB(T_{query} + T_{int})) = O(SMB(T_{query} + T_{int})). \quad (5.4)$$

In the next section, we proceed to analyze the asymptotic complexity of T_{int} for the surveyed methods, which is the intersection at the leaf node level of the $closestIntersection$ function.

5.5.2 Asymptotic computational complexity

We analyze the *worst case* behavior of the surveyed methods' intersection algorithms within the acceleration structure leaf node. In this analysis, we consider the effect of the number of neighboring points or other primitives K, evaluation points k in regular sampling methods, and iterations I in iterative refinement algorithms.

At the leaf node level, intersection algorithms presented by the methods fall into six categories: splat intersections [24, 34, 41], MLS surface intersections [2, 58, 59], implicit surface intersections [1, 25, 29, 33, 63], projection based intersection evaluation [55] [64], height-difference based intersection [14], and k-NN intersection [15].

Splat intersection

The general splat intersection algorithm consists of the following procedure: within a neighborhood of a leaf node L, with K number of splats s, intersect all splats s with a ray $\boldsymbol{r}$ and produce a weighted average of intersection locations $\boldsymbol{x}$, normals $\boldsymbol{n}$ and other attributes based on the weighted distance from the corresponding splat centers. This amounts to an execution time of $O(K)$.

MLS surface intersection

The general MLS surface intersection algorithm can be seen in Algorithm 3. On lines 2 and 3, the MLS surface is constructed as described in Section 5.1.2, with the projected point being the bounding sphere center $\boldsymbol{p}$. On line 4, the constructed polynomial g, which is a reduced second order polynomial of the form $ax^2 + bx^2 + cxy + d$, is intersected by the ray (Section 3.4). The exit conditions are on lines 5 and 11. Lines 5 to 8 terminate the while-loop and returns the outputs with no value. Lines 11 to 14 return the intersection point $\boldsymbol{p}'$ and the corresponding normal $\boldsymbol{n}'$ if the intersection point's projection distance to the newly evaluated local MLS surface (line 10) is smaller than the threshold value ϵ.

```
Data: Point $\boldsymbol{p}$ with bounding sphere radius $r$ and $K$ number of neighboring points
      $\boldsymbol{p}_i$, ray $\boldsymbol{r}$, threshold $\epsilon$
Result: Intersection point $\hat{\boldsymbol{x}}$ with accompanying attributes, intersection normal $\hat{\boldsymbol{n}}$
1  while true do
2      $\boldsymbol{n}, D \leftarrow solve(\arg\min_{\boldsymbol{n},D} \sum_{i=1}^{K} (\langle \boldsymbol{n}, \boldsymbol{p}_i \rangle - D)^2 \, \theta(\|\boldsymbol{p}_i - \boldsymbol{p}\|))$;
3      $g \leftarrow solve(\arg\min_{g} \sum_{i=1}^{K} (g(x_i, y_i) - \langle \boldsymbol{n}, \boldsymbol{p}_i \rangle)^2 \, \theta(\|\boldsymbol{p}_i - \boldsymbol{p}\|))$;
4      $\boldsymbol{p}' \leftarrow intersectPolynomial(g, \boldsymbol{r})$;
5      if $\boldsymbol{p}' = null$ then
6          $\hat{\boldsymbol{x}} \leftarrow null$;
7          $\hat{\boldsymbol{n}} \leftarrow null$;
8          return;
9      else
10         $\boldsymbol{n}', D' \leftarrow solve(\arg\min_{\boldsymbol{n}',D'} \sum_{i=1}^{K} (\langle \boldsymbol{n}', \boldsymbol{p}_i \rangle - D')^2 \, \theta(\|\boldsymbol{p}_i - \boldsymbol{p}'\|))$;
11         if $\langle \boldsymbol{n}', \boldsymbol{p}' \rangle - D' \leq \epsilon$ then
12             $\hat{\boldsymbol{x}} \leftarrow \boldsymbol{p}'$;
13             $\hat{\boldsymbol{n}} \leftarrow \boldsymbol{n}'$;
14             return;
15         else
16             $\boldsymbol{p} \leftarrow \boldsymbol{p}'$;
17             $\boldsymbol{n} \leftarrow \boldsymbol{n}'$;
18         end
19     end
20 end
```

Algorithm 3: A general MLS surface intersection algorithm utilized in [2, 58, 59].

Otherwise the iterative MLS procedure is continued.

The deciding factors in the time complexity of Algorithm 3 are lines 2, 3 and 10, and the number of iterations of the while-loop in general. Both [2] and [58] report 2 or 3 iterations for any given ray with reasonably chosen threshold ϵ. In [59], the iterative refinement is omitted and the first ray-polynomial intersection query is used. The execution time for $intersectPolynomial$ (line 4) is dependent on the order of the polynomial (see Section 3.4), but not on K. A reduced second order polynomial is used in all methods and the intersection can be solved with Equation (3.13), which is not dependent on K or the other asymptotic variables. Finally, the execution times T_2^3, T_3^3 and T_{10}^3 relate to solving the local MLS surface. As discussed in Section 5.1.2, the complexity of the fitting problem grows linearly to the number of neighbors K considered in the fit, thus the computational complexity for the whole algorithm is concluded to be $O(IK)$.

Implicit surface intersection algorithms

The implicit surface octree method used by Goradia et al. [25] and Kashyap et al. [33] is described in Algorithm 4. The first evaluation point is initialized to the octree leaf node entering intersection point (line 2) with accompanying implicit function value (line 3). The for-loop (line 5) sub-divides the ray within the leaf node to k equally spaced evaluation points (line 4). An intersection is reported if the values of the implicit function at the current evaluation point (lines 6 and 7) and the previous evaluation point have different signs (line 8). Otherwise, the previous evaluation point and implicit function value are set to the current values (lines 13 and 14) and the evaluation the looping is continued. The function $Lerp$ (line 9) linearly interpolates the intersection point based on the implicit function values at the current and previous evaluation points by Equation (3.20). Similarly, the $triLerp$ function (lines 3 and 7) trilinearly interpolates the implicit function values in the corners of the octree leaf based on the relative position of the evaluation point inside the leaf (Section 3.6.1). In the worst case, no intersection is found within the leaf node and a miss is reported by returning $null$ values as output. This yields k iterations of the for loop giving a $O(k)$ worst case complexity with respect to the number of evaluation points k.

```
Data: Eight octree leaf node corner points $\boldsymbol{p}[1-8]$ with implicit functions $f[1-8]$
      and normals $\boldsymbol{n}[1-8]$, ray $\boldsymbol{r}$ with entering $t_{min}$ and exiting $t_{max}$ intersections
      with octree leaf node, $k$ marching points
Result: Intersection point $\hat{\boldsymbol{x}}$ with accompanying attributes, intersection normal $\hat{\boldsymbol{n}}$
1  $t_{prev} \leftarrow t_{min}$;
2  $\boldsymbol{x}_{prev} \leftarrow \boldsymbol{r}(t_{prev})$;
3  $f_{prev} \leftarrow triLerp(\boldsymbol{x}_{prev}, \boldsymbol{p}[1-8], f[1-8])$;
4  $step \leftarrow (t_{max} - t_{min})/k$;
5  for $t \leftarrow t_{min}$to $t_{max}$ step $step$ do
6      $\boldsymbol{x}_{curr} \leftarrow \boldsymbol{r}(t)$;
7      $f_{curr} \leftarrow triLerp(f[1-8], \boldsymbol{x}_{curr}, \boldsymbol{p}[1-8])$;
8      if $f_{curr} \cdot f_{prev} \leq 0$ then
9          $\hat{\boldsymbol{x}} \leftarrow Lerp(\boldsymbol{x}_{prev}, \boldsymbol{x}_{curr}, f_{prev}, f_{curr})$;
10         $\hat{\boldsymbol{n}} \leftarrow triLerp(\boldsymbol{n}[1-8], \hat{\boldsymbol{x}}, \boldsymbol{p}[1-8])$;
11         return;
12     else
13         $\boldsymbol{x}_{prev} \leftarrow \boldsymbol{x}_{curr}$;
14         $f_{prev} \leftarrow f_{curr}$;
15     end
16 end
17 $\hat{\boldsymbol{x}} \leftarrow null$;
18 $\hat{\boldsymbol{n}} \leftarrow null$;
```

Algorithm 4: An implicit surface marching algorithm used in [25, 33].

Wald and Seidel [63] also utilize a regularly sampled implicit surface intersection approach which is shown in Algorithm 5. The method regularly samples the ray within the leaf node k times (inner for loop), but when a sign change is recognized, the interval between t_{prev} and t_{curr} is again sampled at k evaluation points (outer while loop). This iterative procedure is resumed until the implicit function value is within an ϵ threshold of zero at some evaluation point (lines 12–15) yielding up to I iterative steps. Furthermore, instead of interpolating precalculated leaf node corner values as in Algorithm 4, a distance weighted average of all K number of point locations $\boldsymbol{p}_i$ and corresponding normals $\boldsymbol{n}_i$ (line 10) within the leaf node are calculated with Equation (3.22). The weighting function θ used in the method is defined as $\theta(a) = max(1 - a, 0)$. The distance weighted averages $\boldsymbol{p}_{curr}$ and $\boldsymbol{n}_{curr}$ are used to construct a local plane (line 10) defined by the implicit function f_{curr} (see Equation (3.11)). In the worst case, the implicit function has to be evaluated for all evaluation points at each iteration step in order find the interval where the sign changes yielding a complexity of $O(kIK)$.

```
Data: Ray $\boldsymbol{r}$ with entering $t_{min}$ and exiting $t_{max}$ intersection bounds with the leaf
      node, $\boldsymbol{p}_i$ and their normals $\boldsymbol{n}_i$ within the leaf node, $k$ marching points, $\epsilon$
      threshold, $\theta$ weighting function
Result: Intersection point $\hat{\boldsymbol{x}}$ with accompanying attributes, intersection normal $\hat{\boldsymbol{n}}$
1  while true do
2      $t_{prev} \leftarrow t_{min}$;
3      $\boldsymbol{x}_{prev} \leftarrow \boldsymbol{r}(t_{prev})$;
4      $\boldsymbol{p}_{prev}, \boldsymbol{n}_{prev} \leftarrow weightedAvg(\boldsymbol{x}_{prev}, \boldsymbol{p}_i, \boldsymbol{n}_i, \theta)$;
5      $f_{prev} \leftarrow dot(\boldsymbol{n}_{prev}, (\boldsymbol{x}_{prev} - \boldsymbol{p}_{prev})$;
6      $step \leftarrow (t_{max} - t_{min})/k$;
7      $\boldsymbol{x} \leftarrow \boldsymbol{r}(t_{min})$;
8      for $t \leftarrow t_{min}$to $t_{max}$ step $step$ do
9          $\boldsymbol{x}_{curr} \leftarrow \boldsymbol{r}(t)$;
10         $\boldsymbol{p}_{curr}, \boldsymbol{n}_{curr} \leftarrow weightedAvg(\boldsymbol{x}_{curr}, \boldsymbol{p}_i, \boldsymbol{n}_i, \theta)$;
11         $f_{curr} \leftarrow dot(\boldsymbol{n}_{curr}, (\boldsymbol{x}_{curr} - \boldsymbol{p}_{curr})$;
12         if $\|f_{curr}\| < \epsilon$ then
13             $\hat{\boldsymbol{x}} \leftarrow \boldsymbol{x}_{curr}$;
14             $\hat{\boldsymbol{n}} \leftarrow \boldsymbol{n}_{curr}$;
15             return;
16         else if $f_{curr} \cdot f_{prev} < 0$ then
17             $t_{min} \leftarrow t_{prev}$;
18             $t_{max} \leftarrow t$;
19             break ;          /* Goes back to beginning of while loop */
20
21         else if $t = t_{max}$ then
22             $\hat{\boldsymbol{x}} \leftarrow null$;
23             $\hat{\boldsymbol{n}} \leftarrow null$;
24             return;
25         else
26             $t_{prev} \leftarrow t$;
27             $\boldsymbol{x}_{prev} \leftarrow \boldsymbol{x}_{curr}$;
28             $f_{prev} \leftarrow f_{curr}$;
29         end
30     end
31 end
```

Algorithm 5: An iterative implicit surface intersection algorithm used in [63]

The Newton's method variant described in Equation (3.24) is utilized by Adams et al. [1]. Algorithm 6 describes the iterative procedure for implicit surface intersection evaluation. The initial evalution point $\boldsymbol{x}$ is set to the entering intersection point $\boldsymbol{r}(t_{min})$ to the leaf node (line 1). Distance weighted averages of point locations $\boldsymbol{p}$ and normals $\boldsymbol{n}$ are calculated based on the point locations $\boldsymbol{p}_i$ and normals $\boldsymbol{n}_i$ within the leaf node (line 3). An implicit function f is constructed to define a plane with point $\boldsymbol{p}$ on the plane and normal $\boldsymbol{n}$ as the plane normal (line 4). If the value of the implicit function f at evaluation point $\boldsymbol{x}$ is within threshold distance ϵ of zero, then an intersection is reported at $\boldsymbol{x}$ with surface normal $\boldsymbol{n}$ (lines 5–8). Otherwise, the plane is intersected by the ray (see Equation (3.12)) and the intersection point is used as the new evaluation point for the next iteration step (lines 10 and 16). However, the intersection point must be within the leaf node bounds, otherwise the no intersection is reported within the leaf node (lines 11–14). In the worst case, I iterative steps have to be taken until convergence and the weighted average for K number of points within the leaf node has to be evaluated yielding a complexity of $O(IK)$.

Hubo et al. [28] also use the Newton's method variant of the iterative implicit surface intersection described in Algorithm 6. For the weighting function θ, the method uses a Gaussian weighting function (see Equation (3.15)) with zero mean and a variance σ^2 to control the smoothness of the resulting surface. As discussed previously, this yields a worst case complexity of $O(IK)$.

In the improved method by Hubo et al. [29], the implicit surface construction is varied slightly from Algorithm 6. Instead of constructing a single implicit surface from all the points within the leaf node, the method uses several precalculated implicit surface patches inside the node and intersects them separately. Each patch is implicitly defined by the distance weighted average of the subset of points within the leaf node with Gaussian weighting. If an intersection is found with more than one patch, the intersection point values are averaged based on the Gaussian weighted distance from the intersected patches respective centers. The method of intersecting the patches is the Newton's method variant in Algorithm 6. This gives a worst case complexity of $O(IK)$.

Point projection based intersection

Schaufler and Jensen [55] use a projection based intersection described in Algorithm 7. The entering intersection point $\boldsymbol{r}(t_{min})$ to the leaf node is used as a projection point $\boldsymbol{x}$ along the ray (line 1). Each points' $\boldsymbol{p}_i$ distance to the projection point $\boldsymbol{x}$ is calculated along the points' normals $\boldsymbol{n}_i$ and if all distances are larger than the ray cylinder radius d, then the ray misses all points within the leaf node (lines 2–5). Otherwise, the weighted average of locations and normals of all point within the ray radius are calculated using the points' projected distances and they are used as the intersection point $\hat{\boldsymbol{x}}$ and the corresponding normal $\hat{\boldsymbol{n}}$. The factors determining the complexity in the algorithm are lines 2 and 7 which iterate through all K number of points once giving a worst case complexity of $O(K)$ in total.

A variation of Algorithm 7 is used by Wand and Straßer [64]. The rays are represented by cones instead of cylinders and two radii u_d and v_d are calculated along orthonormal vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ (see Theorem 3.9) orthogonal to the ray $\boldsymbol{r}(t) = \boldsymbol{o} + \boldsymbol{d}t$. The radii u_d and v_d change linearly as a function of the ray parameter t. A single point $\boldsymbol{p}_i$ inside the leaf node is projected onto ray coordinates $\boldsymbol{u}$, $\boldsymbol{v}$, and $\boldsymbol{d}$ by $u_{\boldsymbol{p}_i} = (\boldsymbol{o} - \boldsymbol{p}_i) \cdot \boldsymbol{u}$, $v_{\boldsymbol{p}_i} = (\boldsymbol{o} - \boldsymbol{p}_i) \cdot \boldsymbol{v}$,

```
Data: Ray $\boldsymbol{r}$ with entering $t_{min}$ and exiting $t_{max}$ intersection bounds with the leaf
      node, points $\boldsymbol{p}_i$ and their normals $\boldsymbol{n}_i$ within the leaf node, $\theta$ weighting function
Result: Intersection point $\hat{\boldsymbol{x}}$ with accompanying attributes, intersection normal $\hat{\boldsymbol{n}}$
$\boldsymbol{x} \leftarrow \boldsymbol{r}(t_{min})$;
while true do
    $\boldsymbol{p}, \boldsymbol{n} \leftarrow weightedAvg(\boldsymbol{x}, \boldsymbol{p}_i, \boldsymbol{n}_i, \theta)$;
    $f(\boldsymbol{y}) := dot(\boldsymbol{n}, \boldsymbol{y} - \boldsymbol{p})$;
    if $\|f(\boldsymbol{x})\| < \epsilon$ then
        $\hat{\boldsymbol{x}} \leftarrow \boldsymbol{x}$;
        $\hat{\boldsymbol{n}} \leftarrow \boldsymbol{n}$;
        return;
    else
        $t \leftarrow Intersect(\boldsymbol{r}, f(\boldsymbol{y}))$;
        if $t > t_{max}$ or $t < t_{min}$ then
            $\hat{\boldsymbol{x}} \leftarrow null$;
            $\hat{\boldsymbol{n}} \leftarrow null$;
            return;
        else
            $\boldsymbol{x} \leftarrow \boldsymbol{r}(t)$;
        end
    end
end
```

Algorithm 6: An iterative implicit surface intersection algorithm used in [1, 28].

and $t = (\boldsymbol{o} - \boldsymbol{p}_i) \cdot \boldsymbol{d}$, respectively. The weight of the point $\theta(\boldsymbol{p}_i)$ is then calculated as the distance of the projected point $(u_{\boldsymbol{p}_i}, v_{\boldsymbol{p}_i})$ in the ray coordinates weighted by a gaussian function (see Equation (3.15)) with the ray radii u_d^2 and v_d^2 as the variance σ^2. Final intersection location is at the weighted average of the locations of the points inside the ray cone and the corresponding attributes are averaged with the same weighting. Similar to Algorithm 7, the points within the leaf node are iterated through once to calculate the weighted average of attributes which gives a worst case complexity of $O(K)$.

A height-difference implicit surface intersection

Cai et al. [14] assume that radii r_i and normals $\boldsymbol{n}_i$ are present for the points $\boldsymbol{p}_i$ within the leaf node. Thus, each point defines an oriented circular splat and is bounded by a sphere with $\boldsymbol{p}_i$ center and r_i radius. All K number of bounding spheres are queried for intersections (see Equation (3.14)). The middle point of the two intersections for each sphere is stored as an initial iteration point $\boldsymbol{r}(t_{i0})$. Spheres with an initial iteration parameter $t_{i0} \notin [t_{min}, t_{max}]$ are discarded. If all spheres are discarded or missed by the ray, an intersection was not found within the leaf node. Otherwise, the algorithm continues with the set of sphere intersection middle points into iterative intersection refinement.

The iterative intersection refinement is done to all valid sphere intersections starting from the closest, i.e., from the smallest t_i. An average normal direction $\bar{\boldsymbol{n}}_j$ is calculated from

```
Data: Ray $\boldsymbol{r}$ with entering $t_{min}$ and exiting $t_{max}$ intersection bounds with the leaf
      node, $d$ ray cylinder radius, points $\boldsymbol{p}_i$ and their normals $\boldsymbol{n}_i$ within the leaf
      node, $\theta_d$ weighting function with cut-off distance at $d$
Result: Intersection point $\hat{\boldsymbol{x}}$ with accompanying attributes, intersection normal $\hat{\boldsymbol{n}}$
$\boldsymbol{x} \leftarrow \boldsymbol{r}(t_{min})$;
if $\forall i : \|dot(\boldsymbol{n}_i, \boldsymbol{x} - \boldsymbol{p}_i)\| > d$ then
    $\hat{\boldsymbol{x}} \leftarrow null$;
    $\hat{\boldsymbol{n}} \leftarrow null$;
    return;
else
    $\boldsymbol{p}, \boldsymbol{n} \leftarrow weightedAvg(\boldsymbol{x}, \boldsymbol{p}_i, \boldsymbol{n}_i, \theta_d)$;
    $\hat{\boldsymbol{x}} \leftarrow \boldsymbol{p}$;
    $\hat{\boldsymbol{n}} \leftarrow \boldsymbol{n}$;
    return;
end
```

Algorithm 7: A ray cylinder point projection algorithm used in [1].

all point splat normals $\boldsymbol{n}_i$ that are within radius r_i distance of the current iteration point $\boldsymbol{r}(t_{ij})$. The iteration point $\boldsymbol{r}(t_{ij})$ is projected to the splat plane of points at r_i distance along the average normal $\bar{\boldsymbol{n}_j}$. The distance of the projection defines a height difference from the current iteration point to the near splats. Each height difference is given a weight calculated by the squared distance of the projected point to the respective splat's center $\boldsymbol{p}_i$ normalized by the splat radius r_i. If the projection point is outside the splat radius, then the weight is set to zero.

Finally, a weighted average of the height differences is calculated. If the averaged height difference is within a threshold ϵ of zero, the current iteration point $\boldsymbol{r}(t_{ij})$ and a weighted average of splat normals are returned as the intersection. Otherwise, t_{ij} is incremented or decremented by the averaged height difference and the iteration continues until convergence or divergence by exiting the bounding sphere of the splats.

The algorithm goes through all K point splats once and iterates maximally I times. Because there is no k-NN structure present, the algorithm has to query all point splats inside the leaf node to find the ones considered in this iteration step. This gives K more queries for each iteration. In total, a worst case complexity of $O(KIK) = O(IK^2)$ is established.

A k-NN point intersection

A k-NN based method by Cai et al. [15] intersects leaf node points without point normal information. Starting from the entering intersection at t_0 to the leaf node, nearest neighbors are queried from a precalculated k-NN structure at predefined intervals δt until at least one neighbor is found. The distance d_{near} to the nearest point $\boldsymbol{p}_{near}$ is calculated and the next interval is set to $\delta = \frac{d_i}{2}$. If a predefined K number of points is found, the center location $\bar{\boldsymbol{p}} = \sum \boldsymbol{p}_i / k$ of the k nearest points is calculated. Then the distance d_{far}

to the farthest point $\boldsymbol{p}_{far}$ from the average location $\bar{\boldsymbol{p}}$ is calculated and if the distance from the current evaluation point on the ray to $\bar{\boldsymbol{p}}$ is larger than d_{far}, the next interval is taken. Also, if the distance is larger than the distance to any points' $\boldsymbol{p}_i$ bounding radius, the next interval is again taken. Otherwise, the algorithm proceeds to surface orientation evaluation.

Triangles $\delta(\bar{\boldsymbol{p}}, \boldsymbol{p}_i, \boldsymbol{p}_j)$ are drawn between all possible points. The area weighted average of the triangle normals is calculated and used as the normal $\bar{\boldsymbol{n}}$ at the current evaluation point. Projected distance along the normal $\bar{\boldsymbol{n}}$ from the current evaluation point to $\boldsymbol{p}_i$ is calculated as h. If h/d_{far} is smaller than a positive threshold ϵ, an intersection is at the current evaluation point with normal $\bar{\boldsymbol{n}}$. Otherwise, the new interval is set to $\delta t = h/2$ and the next interval taken. In the worst case, there are up to I iterative steps each having $(K-1)K/2 = O(K^2)$ triangle evaluations resulting in $O(IK^2)$ complexity.

Summary

The asymptotic computational complexities of the intersection methods at the leaf node level are summarized in Table 5.5. In general, the computational complexity of the intersection methods varies depending on the iterative or non-iterative nature of the algorithms. Almost all methods need to go through the points inside the leaf node at least ones yielding $O(K)$ complexity for the asymptotically most efficient methods. Similar asymptotic efficiency, namely $O(k)$ complexity, is achieved when regular sampling is used to find the intersection.

5.6 Limitations of the analysis

For the acceleration structure construction complexity analysis, we rely solely on reported asymptotic complexities in other publications. These analyses may be subject to limitations and preconditions that might not be met by the surveyed methods' algorithms. The choice to use the reported complexities was done because not all of the surveyed methods described how they constructed their acceleration structures. Reviewing the compatibility of the acceleration structure construction algorithms is out of scope in this thesis.

The generic ray tracing algorithm (Algorithm 1) and the rendering equation evaluator algorithm (Algorithm 2) are simplified versions of algorithms that might be used in an actual production grade ray tracer. Advanced ray tracing techniques discussed in Chapter 4 are not present in either of the algorithms. However, to the best of the author's knowledge injecting the following ray tracing techniques into the algorithms doesn't affect the asymptotic complexity: shadow rays, refraction rays, importance sampling, and different camera models.

Other preprocessing methods discussed previously were not analyzed for asymptotic

Table 5.5. *Asymptotic complexity of the intersection within the leaf node is summarized for the surveyed methods. The table shows the worst case asymptotic computational complexities with the asymptotic variables K (number of neighboring points), k (number of regularly sampled intervals), and I (number of steps in iterative intersection refinement). A reduced analysis based on the most determining input variables is also presented.*

Method	**Intersection complexity**
Schaufler and Jensen 2000 [55]	$O(K)$
Adamson and Alexa 2003 [2]	$O(IK)$
Wand and Straßer 2003 [64]	$O(K)$
Adams et al. 2005 [1]	$O(IK)$
Wald and Seidel 2005 [63]	$O(kIK)$
Tejada et al. 2006 [58]	$O(IK)$
Hubo et al. 2006 [28]	$O(IK)$
Hubo et al. 2007 [29]	$O(IK)$
Tejada et al. 2007 [59]	$O(K)$
Linsen et al. 2007 [41]	$O(K)$
Kashyap et al. 2010 [34]	$O(K)$
Goradia et al. 2010 [24]	$O(K)$
Kashyap et al. 2010 [33]	$O(k)$
Goradia et al. 2011 [25]	$O(k)$
Cai et al. 2011 [14]	$O(IK^2)$
Cai et al. 2015 [15]	$O(IK^2)$

complexity. It is plausible that in the asymptotic sense they would dominate the acceleration structure construction time. Furthermore, the asymptotic analysis of the rendering time complexity (Table 6.1) assumes static point cloud models across rendered frames. This means that dynamic updates for the acceleration structures or other preprocessing methods are not accounted for in the case of animated geometry. Even though the method by Adams et al. [1] supports deformation of the point cloud geometry, the analysis assumes a static point cloud.

The pseudocode algorithms presented in this chapter serve as guidelines for the asymptotic complexity analysis and are not replicates of the original surveyed methods. The algorithms do not depict exact implementation details of the methods and they should not be used as such when implementing the surveyed methods. The reader is advised to familiarize themselves with the original publication before implementing the surveyed methods.

6 RESULTS

The surveyed methods and the identified categories are summarized in Table 5.1. Seven major categories were established. Real-time capabilities were indicated in nine of the methods. MLS surface, implicit surface, and splat definitions were the three significant ways of representing the points for intersections. Furthermore, a minor category of projecting points onto the ray was also established.

Many of the methods used iterative refinement to converge to an exact intersection. A regularly sampled interval, was also a common method to detect an intersection.

Based on the survey, acceleration structures and preprocessing methods were utilized in almost all of the ray tracing algorithms present. The usage of acceleration structures in ray tracing in general is very common, as stated in several relevant books on computer graphics [4, 30, 49]. Out of the other preprocessing methods, the most popular ones were the construction of a k-NN structure and the generation of splats (Table 5.4). If splats weren't generated by the surveyed method themselves, most of the methods still needed point radii and normals as a prerequisite for ray tracing (Table 5.3).

Results of the asymptotic complexity analysis for the surveyed methods are presented in Table 6.1. The biggest differences at the leaf node level intersections were due to the iterative nature of intersection algorithms. However, many methods reported that the number of iterations I needed for convergence was small enough not to affect the execution time. Other methods also used regularly spaced evaluation points to approximate the intersection point, which was present as the number of marching intervals k. The common factor in all leaf node level intersections was the number of neighboring points K within the leaf node.

Based on the most determining asymptotic variables, namely the number of pixels M and points N, there was no significant difference between the methods at rendering time. The difference between the two methods exhibiting $O(MN)$ complexity and the rest of the methods with $O(M \log N)$ complexity, was whether they used ray tracing with acceleration or rasterizing without acceleration to find the closest intersection in asymptotic behavior at rendering time was whether the method used ray tracing and acceleration to find the closest intersection.

Compared to the mesh generation with Delaunay triangulation, the asymptotic complexity difference depends on the complexity of preprocessing. All but three of the surveyed methods use some preprocessing (Table 5.4) and the analysis of the preprocessing methods wasn't done. However, the methods that didn't preprocess the point cloud have trivially asymptotic complexity of $O(1)$ before acceleration structure construction compared to triangle mesh construction with asymptotic complexity of $O(N^3)$ in the worst case and $O(N)$ in the best case. Combining this with the acceleration structure construction complexity of $O(N \log N)$, the surveyed methods have a better complexity of $O(N \log N)$ compared to $O(N^3)$ in the worst case for the Delaunay triangulation.

Table 6.1. *Asymptotic complexity of the rendering time of a single frame. The table shows the worst case asymptotic computational complexities with all asymptotic variables M (number of pixels), N (number of point cloud points), S (spp), B (number of recursive ray bounces), K (number of neighboring points), k (number of regularly sampled intervals), and I (number of steps in iterative intersection refinement). A reduced analysis based on the most determining input variables is also presented.*

Method	**Asymptotic complexity**	**Reduced asymptotic complexity**
Schaufler and Jensen 2000 [55]	$O(SMB(\log(N) + K))$	$O(M \log N)$
Adamson and Alexa 2003 [2]	$O(SMB(\log(N) + IK))$	$O(M \log N)$
Wand and Straßer 2003 [64]	$O(SMB(\log(N) + K))$	$O(M \log N)$
Adams et al. 2005 [1]	$O(SMB(\log(N) + IK))$	$O(M \log N)$
Wald and Seidel 2005 [63]	$O(SMB(\log(N) + kIK))$	$O(M \log N)$
Tejada et al. 2006 [58]	$O(MNIK + SBNIK)$	$O(MN)$
Hubo et al. 2006 [28]	$O(SMB(\log(N) + IK))$	$O(M \log N)$
Hubo et al. 2007 [29]	$O(SMB(\log(N) + IK))$	$O(M \log N)$
Tejada et al. 2007 [59]	$O(MNK + SBNK)$	$O(MN)$
Linsen et al. 2007 [41]	$O(SMB(\log(N) + K))$	$O(M \log N)$
Kashyap et al. 2010 [34]	$O(SMB(\log(N) + K))$	$O(M \log N)$
Goradia et al. 2010 [24]	$O(SMB(\log(N) + K))$	$O(M \log N)$
Kashyap et al. 2010 [33]	$O(SMB(\log(N) + k))$	$O(M \log N)$
Goradia et al. 2011 [25]	$O(SMB(\log(N) + k))$	$O(M \log N)$
Cai et al. 2011 [14]	$O(SMB(\log(N) + IK^2))$	$O(M \log N)$
Cai et al. 2015 [15]	$O(SMB(\log(N) + IK^2))$	$O(M \log N)$

7 CONCLUSIONS

In this thesis, direct point cloud ray tracing methods were surveyed and asymptotic complexity analysis was used to compare the methods' algorithms for intersection testing. Three main local surface categories for point cloud ray tracing were identified, namely the moving least squares (MLS) surface, implicit surface and splat definitions. A minor category of point projection based intersection evaluation was also established.

The asymptotic complexity analysis concluded that the main differences between the methods were at the leaf node level and were due to iterative or regularly sampled intersection evaluations. The number of pixels and point cloud points dominated the asymptotic complexity and showed that the main difference between the algorithms on a higher level was in the choice of using ray tracing and acceleration structures compared to rasterizing without acceleration structures.

Real-time capabilities of direct point cloud ray tracing were demonstrated by Wald and Seidel and in several other surveyed methods (see Table 5.1). The best methods in asymptotic computational complexity achieved either $O(K)$ or $O(k)$ complexity with respect to the number of points K within the leaf node and regular sampling intervals k. All projection based [55, 64] and splat intersection methods [24, 34, 41], and some implicit surface methods [25, 33] achieved the best asymptotic complexities.

From the perspective of the whole rendering pipeline, using an intersection acceleration structure in point cloud ray tracing is beneficial if rasterization techniques are not utilized. As both global surface reconstruction and direct ray tracing methods benefit from this, the additional utility from direct ray tracing depends on other mandatory preprocessing. If point normals and radii are present as attributes, many of the surveyed methods don't need additional preprocessing making them viable compared to global surface reconstruction.

The local MLS surface method by Adamson and Alexa [2] only needs point locations for accurate intersections making it the method of choice for point clouds without additional attributes. We conclude that in the worst case of ray tracing a single frame, Adamson and Alexa have a $O((N+M)\log N)$ time complexity which is better than the $O(N^2+M\log N)$ complexity of mesh reconstruction.

Compared to global point cloud surface reconstruction, direct ray tracing could be applied to photorealistic visualization of point cloud data that is generated and streamed in real-time. Furthermore, it is plausible to use direct point cloud ray tracing for partial or incomplete point clouds because only local points have to be considered in the intersection testing. Combining this with the methods that utilize point cloud compression [25, 28, 29, 33], an application in memory bound point cloud visualization systems could be feasible.

The comparison of computational efficiency of the surveyed methods was limited to asymptotic analysis in this thesis. Implementations on current hardware weren't tested and comparing the reported efficiency in the publications wasn't feasible, due to publication dates spanning two decades.

In the future, it may be interesting to implement and compare the absolute efficiency of the surveyed methods on modern desktop and mobile hardware. Comparing the visual quality of the methods can give insight into the specific benefits and challenges in the identified method categories. Additionally, different surface reconstruction methods, such as mesh reconstruction, could be compared visually and in absolute computational efficacy to direct point cloud ray tracing.

REFERENCES

[1] B. Adams, R. Keiser, M. Pauly, L. Guibas, M. Gross, and P. Dutré. Efficient Raytracing of Deforming Point-Sampled Surfaces. In: *Computer Graphics Forum* 24.3 (2005), pp. 677–684. DOI: 10.1111/j.1467-8659.2005.00892.x.

[2] A. Adamson and M. Alexa. Ray tracing point set surfaces. English. In: *Proceedings of the Shape Modeling International*. 2003, pp. 272–279. DOI: 10.1109/SMI.2003.1199627.

[3] T. Akenine-Möller, E. Haines, and N. Hoffman. *Real-time rendering*. 3rd edition. Taylor & Francis Ltd, 2008, 1025 p. ISBN: 9781568814247.

[4] T. Akenine-Möller, E. Haines, and N. Hoffman. *Real-time rendering*. 4th edition. Taylor & Francis Ltd, 2018, 1198 p. ISBN: 9781138627000.

[5] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, and C. Silva. Point Set Surfaces. In: *Proceedings of the Conference on Visualization '01*. 2001, pp. 21–28. URL: http://dl.acm.org/citation.cfm?id=601671.601673.

[6] N. Amenta, M. Bern, and M. Kamvysselis. A new Voronoi-based surface reconstruction algorithm. In: SIGGRAPH '98. ACM, 1998, pp. 415–421. DOI: 10.1145/280814.280947.

[7] T. Ashton and J. Cantarella. A fast octree-based algorithm for computing ropelength. In: *Physical And Numerical Models In Knot Theory: Including Applications to the Life Sciences*. World Scientific, 2005, pp. 323–341. DOI: 10.1142/9789812703460_0017.

[8] K. Atkinson. *An introduction to numerical analysis*. 2. ed. John Wiley & Sons, 1989. ISBN: 0471624896.

[9] Y. Bai and D. Wang. On the Comparison of Trilinear, Cubic Spline, and Fuzzy Interpolation Methods in the High-Accuracy Measurements. In: *IEEE Transactions on Fuzzy Systems* 18.5 (2010), pp. 1016–1022. DOI: 10.1109/TFUZZ.2010.2064170.

[10] E. Barbeau. *Polynomials*. Problem Books in Mathematics. Springer, 2003. ISBN: 9780387406275.

[11] M. Berger, A. Tagliasacchi, L. M. Seversky, P. Alliez, G. el Guennebaud, J. A. Levine, A. Sharf, and C. Silva. A Survey of Surface Reconstruction from Point Clouds. In: *Computer Graphics Forum* 36.1 (2017), pp. 301–329. DOI: 10.1111/cgf.12802.

[12] F. Bijma. *Introduction to Mathematical Statistics*. Amsterdam University Press, 2017, 372. ISBN: 9048536111.

[13] B. Burley, D. Adler, M. J.-Y. Chiang, H. Driskill, R. Habel, P. Kelly, P. Kutz, Y. K. Li, and D. Teece. The design and evolution of disney's hyperion renderer. In: *ACM Transactions on Graphics (TOG)* 37.3 (2018), pp. 33:1–33:21. DOI: 10.1145/3182159.

[14] P. Cai, D. Kong, S. Wang, and B. Yin. A Height-difference-based Ray Tracing of Point Models. In: *Proceedings of the 10th International Conference on Virtual Reality Continuum and Its Applications in Industry*. ACM, 2011, pp. 91–98. DOI: 10.1145/2087756.2087768.

[15] P. Cai, D. Kong, S. Wang, B. Yin, and Y. Huo. A K-neighbor-based Ray Tracing of Point Clouds. In: *Journal of Information & Computational Science* 12.13 (2015), pp. 4929–4941.

[16] Z.-Q. Cheng, Y.-Z. Wang, B. Li, K. Xu, G. Dang, and S.-Y. Jin. A Survey of Methods for Moving Least Squares Surfaces. In: *Proceedings of the Fifth Eurographics / IEEE VGTC Conference on Point-Based Graphics*. 2008, pp. 9–23. DOI: 10.2312/VG/VG-PBG08/009-023.

[17] P. Christensen, J. Fong, J. Shade, W. Wooten, B. Schubert, A. Kensler, S. Friedman, C. Kilpatrick, C. Ramshaw, M. Bannister, B. Rayner, J. Brouillat, and M. Liani. RenderMan: An Advanced Path-Tracing Architecture for Movie Rendering. In: *ACM Transactions on Graphics (TOG)* 37.3 (2018), pp. 30:1–30:21. DOI: 10.1145/3182162.

[18] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2009, 1292 p. ISBN: 9780262033848. URL: https://dl.acm.org/citation.cfm?id=1614191.

[19] Z. Cvetkovski. *Inequalities: theorems, techniques and selected problems*. Springer-Verlag, 2012. ISBN: 9783642237911. DOI: 10.1007/978-3-642-23792-8.

[20] J. Dieudonné. *Foundations of modern analysis*. Academic Press, 1969, 387 p. ISBN: 978-0122155505.

[21] E. Dougherty. *Probability and statistics for the engineering, computing, and physical sciences*. Prentice-Hall, Inc., 1990. ISBN: 013711995X.

[22] C. Ericson. *Real-time collision detection*. Elsevier, 2005. ISBN: 1558607323.

[23] S. Fortune. Voronoi diagrams and Delaunay triangulations. In: Computing in Euclidean geometry. World Scientific, 1992, pp. 193–233. DOI: 10.1142/9789814355858_0006.

[24] R. Goradia, S. Kashyap, P. Chaudhuri, and S. Chandran. GPU-Based Ray Tracing of Splats. In: *Proceedings of the 18th Pacific Conference on Computer Graphics and Applications*. 2010, pp. 101–108. DOI: 10.1109/PacificGraphics.2010.21.

[25] R. Goradia, S. Kashyap, P. Chaudhuri, and S. Chandran. Tracing specular light paths in point-based scenes. In: *The Visual Computer* 27.12 (2011), 1083–1097. DOI: 10.1007/s00371-011-0654-z.

[26] M. Gross and H. Pfister. *Point-Based Graphics*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007. ISBN: 9780080548821.

[27] J. Hocking and G. Young. *Topology*. Addison-Wesley, 1961, 374 p. ISBN: 978-0486656762.

[28] E. Hubo, T. Mertens, T. Haber, and P. Bekaert. The Quantized kd-Tree: Efficient Ray Tracing of Compressed Point Clouds. In: *Proceedings of the Symposium on Interactive Ray Tracing*. 2006, pp. 105–113. DOI: 10.1109/RT.2006.280221.

[29] E. Hubo, T. Mertens, T. Haber, and P. Bekaert. Self-Similarity-Based Compression of Point Clouds, with Application to Ray Tracing. In: *Proceedings of the Eurographics Symposium on Point-Based Graphics*. 2007. DOI: 10.2312/SPBG/SPBG07/129-137.

[30] J. Hughes, A. V. Dam, M. McGuire, D. Sklar, J. Foley, S. Feiner, and K. Akeley. *Computer graphics: Principles and Practice*. 3. ed. Upper Saddle River, N.J. [u.a.]: Addison-Wesley, 2014. ISBN: 9780321399526.

[31] R. Ivo, C. Vidal, and J. Bento Cavalcante-Neto. A method for clipping splats on sharp edges and corners. In: *The Visual Computer* 28 (2012), pp. 995–1004. DOI: 10.1007/s00371-012-0729-5.

[32] J. Kajiya. The Rendering Equation. In: *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '86. ACM, 1986, pp. 143–150. DOI: 10.1145/15922.15902.

[33] S. Kashyap, R. Goradia, P. Chaudhuri, and S. Chandran. Implicit surface octrees for ray tracing point models. In: *ICVGIP '10 Proceedings of the 7th Indian Conference on Computer Vision, Graphics and Image Processing*. 2010, pp. 227–234. DOI: 10.1145/1924559.1924590.

[34] S. Kashyap, R. Goradia, P. Chaudhuri, and S. Chandran. Real Time Ray Tracing of Point-based Models. In: *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D '10. 2010. DOI: 10.1145/1730804.1730976.

[35] L. Kobbelt and M. Botsch. A survey of point-based techniques in computer graphics. In: *Computers & Graphics* 28.6 (2004), pp. 801–814. DOI: 10.1016/j.cag.2004.08.009.

[36] M. Koskela, K. Immonen, M. Mäkitalo, A. Foi, T. Viitanen, P. Jääskeläinen, H. Kultala, and J. Takala. Blockwise Multi-Order Feature Regression for Real-Time Path-Tracing Reconstruction. In: *ACM Transactions on Graphics (TOG)* 38.5 (2019), pp. 138:1–138:14. DOI: 10.1145/3269978.

[37] E. Kreyszig. *Advanced Engineering Mathematics, 8-th edition*. John Wiley & Sons, 1999. ISBN: 0471154962.

[38] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast BVH construction on GPUs. In: *Computer Graphics Forum*. Vol. 28. 2. 2009, pp. 375–384. DOI: 10.1111/j.1467-8659.2009.01377.x.

[39] O. Lawlor. Algorithms: Rendering. In: *Encyclopedia of Computer Science and Technology* (2017), pp. 116–129.

[40] M. Levoy and T. Whitted. *The use of points as a display primitive*. Citeseer, 1985. URL: https://graphics.stanford.edu/papers/points/.

[41] L. Linsen, K. Müller, and P. Rosenthal. Splat-based Ray Tracing of Point Clouds. In: *Journal of WSCG* 15 (2007), pp. 51–58. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.486.4706.

[42] W. Liu, Y. Cheung, A. Sawant, and D. Ruan. A robust real-time surface reconstruction method on point clouds captured from a 3D surface photogrammetry system. In: *Medical Physics* 43.5 (2016), pp. 2353–2360. DOI: 10.1118/1.4945695.

[43] D. Luenberger. *Optimization by vector space methods*. Wiley, 1969, 326 p. ISBN: 0-471-55359-X.

[44] A. Madansky and H. Alexander. Weighted standard error and its impact on significance testing. In: *The Analytical Group, Inc* (2017).

[45] C. Mallet and F. Bretar. Full-waveform topographic lidar: State-of-the-art. In: *ISPRS Journal of Photogrammetry and Remote Sensing* 64.1 (2009), pp. 1–16. DOI: 10.1016/j.isprsjprs.2008.09.007.

[46] S. Meerits, V. Nozick, and H. Saito. Real-time scene reconstruction and triangle mesh generation using multiple RGB-D cameras. In: *Journal of Real-Time Image Processing* (2017). DOI: 10.1007/s11554-017-0736-x.

[47] M. Nießner, M. Zollhöfer, S. Izadi, and M. Stamminger. Real-time 3D reconstruction at scale using voxel hashing. In: *ACM Transactions on Graphics (TOG)* 32.6 (2013), pp. 169:1–169:11. DOI: 10.1145/2508363.2508374.

[48] A. Papoulis. *Probability & statistics*. eng. Prentice-Hall, 1990, 454 p. ISBN: 0-13-711730-2.

[49] M. Pharr and G. Humphreys. *Physically based rendering*. 2. ed. Burlington, MA: Elsevier, 2010. ISBN: 0123750792.

[50] V. Prasolov. *Polynomials*. Vol. 11. Springer, 2010. ISBN: 3540407146.

[51] R. Preiner, S. Jeschke, and M. Wimmer. Auto Splats: Dynamic Point Cloud Visualization on the GPU. In: *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization*. 2012, pp. 139–148. URL: https://www.cg.tuwien.ac.at/research/publications/2012/preiner_2012_AS/.

[52] G. Reina, P. Gralka, and T. Ertl. A decade of particle-based scientific visualization. In: *The European Physical Journal Special Topics* 227.14 (2019), pp. 1705–1723. DOI: 10.1140/epjst/e2019-800172-4.

[53] M. I. Rosen. Niels Hendrik Abel and Equations of the Fifth Degree. In: *The American Mathematical Monthly* 102.6 (1995), pp. 495–505. DOI: 10.1080/00029890.1995.12004609.

[54] S. Rusinkiewicz and M. Levoy. QSplat: A multiresolution point rendering system for large meshes. In: *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. 2000, pp. 343–352. DOI: 10.1145/344779.344940.

[55] G. Schaufler and H. Jensen. Ray Tracing Point Sampled Geometry. In: *Proceedings of the 11th Eurographics Workshop on Rendering*. Rendering Techniques. 2000, pp. 319–328. DOI: 10.1007/978-3-7091-6303-0_29.

[56] C. Schied, A. Kaplanyan, C. Wyman, A. Patney, C. Chaitanya, J. Burgess, S. Liu, C. Dachsbacher, A. Lefohn, and M. Salvi. Spatiotemporal Variance-guided Filtering: Real-time Reconstruction for Path-traced Global Illumination. In: *Proceedings of High Performance Graphics*. HPG '17. 2017, pp. 2:1–2:12. DOI: 10.1145/3105762.3105770.

[57] J. M. Snyder and A. H. Barr. Ray Tracing Complex Models Containing Surface Tessellations. In: *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '87. ACM, 1987, pp. 119–128. DOI: 10.1145/37401.37417.

[58] E. Tejada, J. Gois, L. Nonato, A. Castelo, and T. Ertl. *Hardware-accelerated Extraction and Rendering of Point Set Surfaces.* 2006. DOI: 10.2312/VisSym/EuroVis06/021-028.

[59] E. Tejada, T. Schafhitzel, and T. Ertl. *Hardware-accelerated point-based rendering of surfaces and volumes*. 2007. URL: https://www.semanticscholar.org/paper/Hardware-accelerated-point-based-rendering-of-and-Tejada-Schafhitzel/1507fcfc496699bd6c5c061f0e4ec31385d06f86.

[60] G. J. Tourlakis. *Lectures in Logic and Set Theory*. Cambridge, UK ; New York: Cambridge University Press, 2003, 575 p. ISBN: 9780511615566. DOI: 10.1017/CBO9780511615566.

[61] I. Wald and V. Havran. On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N). In: *Proceedings of the Symposium on Interactive Ray Tracing*. 2006, pp. 61–69. DOI: 10.1109/RT.2006.280216.

[62] I. Wald, W. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, and P. Shirley. State of the Art in Ray Tracing Animated Scenes. In: *Computer Graphics Forum* (2009), pp. 1691–1722. DOI: 10.1111/j.1467-8659.2008.01313.x.

[63] I. Wald and H.-P. Seidel. Interactive ray tracing of point-based models. In: *Proceedings of the Second Eurographics / IEEE VGTC Conference on Point-Based Graphics*. 2005, pp. 9–16. DOI: 10.2312/SPBG/SPBG05/009-016.

[64] M. Wand and W. Straßer. Multi-Resolution Point-Sample Raytracing. In: *Proceedings of the Graphics Interface 2003 Conference*. CIPS, Canadian Human-Computer Communication Society. 2003, pp. 139–148. DOI: 10.20380/GI2003.17.

[65] T. Whitted. An Improved Illumination Model for Shaded Display. In: *Communications of the ACM* 23.6 (1980), pp. 343–349. DOI: 10.1145/358876.358882.

[66] H. Wilf. *Algorithms and complexity*. 2nd ed. A.K. Peters, 2002, 219 p. ISBN: 978-1568811789.

[67] J. Wu and L. Kobbelt. Optimized Sub-Sampling of Point Sets for Surface Splatting. In: *Computer Graphics Forum* 23.3 (2004), pp. 643–652. DOI: 10.1111/j.1467-8659.2004.00796.x.

[68] M. Zwicker, H. Pfister, J. V. Baar, and M. Gross. Surface splatting. In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. ACM, 2001, pp. 371–378. DOI: 10.1145/383259.383300.