UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

The Line Space - a Directional Data Structure for Ray Tracing Acceleration

DISSERTATION

von

Kevin Keul

Genehmigte Dissertation zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
Fachbereich 4: Informatik
Universität Koblenz-Landau

Vorsitzende/r des Promotionsausschusses: Prof. Dr. Maria A. Wimmer
Vorsitzende/r der Promotionskommission: Prof. Dr. Thomas Burkhardt
Berichterstatter/in und Betreuer/in: Prof. Dr. Stefan Müller
Weitere Berichterstatter: Prof. Dr. Thorsten Grosch

Datum der wissenschaftlichen Aussprache: 04. Dezember 2020

Zusammenfassung

Die Raytracing-Beschleunigung durch dedizierte Datenstrukturen ist schon lange ein wichtiges Thema der Computergrafik. Im Allgemeinen werden dafür zwei unterschiedliche Ansätze vorgeschlagen: räumliche und richtungsbezogene Beschleunigungsstrukturen. Die vorliegende Arbeit stellt einen innovativen kombinierten Ansatz dieser beiden Bereiche vor, welcher weitere Beschleunigung der Strahlenverfolgung ermöglicht. Dazu werden moderne räumliche Datenstrukturen als Basisstrukturen verwendet und um vorberechnete gerichtete Sichtbarkeitsinformationen auf Basis von Schächten innerhalb einer originellen Struktur, dem *Line Space*, ergänzt.

Im Laufe der Arbeit werden neuartige Ansätze für die vorberechneten Sichtbarkeitsinformationen vorgeschlagen: ein binärer Wert, der angibt, ob ein Schacht leer oder gefüllt ist, sowie ein einzelner Vertreter, der als repräsentativer Kandidat die tatsächliche Oberfläche approximiert. Es wird gezeigt, wie der binäre Wert nachweislich in einer einfachen, aber effektiven Leerraum-Überspringungs-Technik (*Empty Space Skipping*) genutzt wird, welche unabhängig von der tatsächlich verwendeten räumlichen Basisdatenstruktur einen Leistungsgewinn beim Raytracing von bis zu 40% ermöglicht. Darüber hinaus wird gezeigt, dass diese binären Sichtbarkeitsinformationen eine schnelle Technik zur Berechnung von weichen Schatten und Umgebungsverdeckung auf der Grundlage von Blockerapproximationen ergeben. Obwohl die Ergebnisse einen gewissen Ungenauigkeitsfehler enthalten, welcher auch dargestellt und diskutiert wird, zeigt sich, dass eine weitere Traversierungsbeschleunigung von bis zu 300% gegenüber der Basisstruktur erreicht wird. Als Erweiterung zu diesem Ansatz wird die repräsentative Kandidatenvorberechnung demonstriert, welche verwendet wird, um die indirekte Lichtberechnung durch die Integration von kaum wahrnehmbaren Bildfehlern signifikant zu beschleunigen. Schließlich werden Techniken vorgeschlagen und bewertet, die auf zweistufigen Strukturen und einer Nutzungsheuristik basieren. Diese reduzieren den Speicherverbrauch und die Approximationsfehler bei Aufrechterhaltung des Geschwindigkeitsgewinns und ermöglichen zusätzlich weitere Möglichkeiten mit Objektinstanziierungen und starren Transformationen.

Alle Beschleunigungs- und Speicherwerte sowie die Näherungsfehler werden gemessen, dargestellt und diskutiert. Insgesamt zeigt sich, dass durch den Line Space eine deutliche Erhöhung der Raytracing-Leistung auf Kosten eines höheren Speicherverbrauchs und möglicher Annäherungsfehler erreicht wird. Die vorgestellten Ergebnisse zeigen damit die Leistungsfähigkeit des kombinierten Ansatzes und eröffnen weitere Möglichkeiten für zukünftige Arbeiten.

Abstract

Ray tracing acceleration through dedicated data structures has long been an important topic in computer graphics. In general, two different approaches are proposed: spatial and directional acceleration structures. The thesis at hand presents an innovative combined approach of these two areas, which enables a further acceleration of the tracing process of rays. State-of-the-art spatial data structures are used as base structures and enhanced by precomputed directional visibility information based on a sophisticated abstraction concept of shafts within an original structure, the *Line Space*.

In the course of the work, novel approaches for the precomputed visibility information are proposed: a binary value that indicates whether a shaft is empty or non-empty as well as a single candidate approximating the actual surface as a representative candidate. It is shown how the binary value is used in a simple but effective empty space skipping technique, which allows a performance gain in ray tracing of up to 40% compared to the pure base data structure, regardless of the spatial structure that is actually used. In addition, it is shown that this binary visibility information provides a fast technique for calculating soft shadows and ambient occlusion based on blocker approximations. Although the results contain a certain inaccuracy error, which is also presented and discussed, it is shown that a further tracing acceleration of up to 300% compared to the base structure is achieved. As an extension of this approach, the representative candidate precomputation is demonstrated, which is used to accelerate the indirect lighting computation, resulting in a significant performance gain at the expense of image errors. Finally, techniques based on two-stage structures and a usage heuristic are proposed and evaluated. These reduce memory consumption and approximation errors while maintaining the performance gain and also enabling further possibilities with object instancing and rigid transformations.

All performance and memory values as well as the approximation errors are measured, presented and discussed. Overall, the Line Space is shown to result in a considerate improvement in ray tracing performance at the cost of higher memory consumption and possible approximation errors. The presented findings thus demonstrate the capability of the combined approach and enable further possibilities for future work.

Acknowledgements

First of all, I would like to express my sincere gratitude to all the people who supported me on my journey with this work. In short: Thank you very much!

The whole list of all these people would be too long for this page. Since some people played a bigger role, I would like to mention them individually. In the first place is my supervising professor Stefan Müller. For years he supported me in my work, showing me further ways of development and training me in more fields than just research. Afterwards, I thank all persons of the computer graphics working group: Brigitte Jung, Anna-Katharina Hebborn, Gerrit Lochmann, Bastian Krayer, Nils Höhner, Nils Lichtenberg, Dominik Grüntjens, Alexander Hug and all others. Special thanks also go to the scientific assistants and co-authors who supported me in my research: Nicolas Klee, Tilman Koß, Felix Schröder, Maximilian Nilles, Paul Lemke and others. Furthermore, I thank all my anonymous reviewers, who didn't always give me nice but definitely always helpful feedback.

Quite apart from my thanks on a professional level, I especially thank my partner, my family, all my friends and the people of my past. The time of my dissertation was not always easy or pleasant. Especially in such times, I knew that I could rely on you and that you could show me the sun even on a rainy day.

Many thanks to all of you.

CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Motivation and Objectives

Rendering of images is one of the major topics in computer graphics. It is commonly divided into two subtopics: rasterization-based image generation in real-time and realistic image synthesis through ray tracing. The latter is especially important with high-quality rendering, as used in movies, advertisement and nowadays with the rise of dedicated ray tracing graphics hardware also in computer games. As such, it has long been an important yet unsolved task to produce realistic images in real-time and its relevance is still rising today. Consequently, a big amount of scientific effort was spent in finding solutions for this task. Among the most researched relevant fields is the acceleration of ray traversal through dedicated data structures, which are used in order to reduce the necessary computation needed for indicating the nearest intersection point for a given ray with the scene. This typically works by recursive spatial subdivision of the scene space and thereby reducing the number of relevant scene primitives for the current ray. While these structures already achieve good results in ray tracing acceleration, they are far from perfect and ongoing research is spent in further improving their potential in traversal acceleration.

While most of the commonly used data structures work in a spatial manner, there are also some approaches based on directional information of the scene. Radiosity systems, for example, use the visibility between all geometrical scene primitives in order to compute the actual light propagation within the scene. But still, they are impractical compared to typical ray tracing systems based on spatial ray traversal acceleration. Apart from that, other directional approaches exist. Ray classification structures were proposed, which use some sort of higher dimension directional volume such as 5D-beams to reduce the amount of potential intersecting scene primitives. This additional dimensionality, however, leads to a significant

increase in needed memory. Moreover, until now the gain in performance is comparable to that of spatial structures, which is why they are less used in ray tracing systems.

Nevertheless, a natural thought in terms of those directional data structures is that memory consumption and initialization time can be traded for runtime performance. Ideally, it is possible to precompute the visibility for every point and every direction in an extensive initialization step, and to use this visibility information for a given point and a given direction with minimal effort, i.e. with just a single query and without further computation overhead during runtime. Obviously, the memory consumption as well as the construction time of this process are immense and hence unfeasible for non-trivial scenes and current hardware. Therefore, the final goal of this work is to find a reasonable discretization of points and directions, such that directional information can be precomputed and used in a meaningful manner in ray traversal acceleration.

This work introduces the *Line Space* (LS) as a directional structure with significantly reduced traversal cost due to the precomputed visibility information. It is integrated into typically used spatial structures and thereby shows a way to combine traditional spatial with directional data structures. It is based on visibility precomputation through an intermediate abstraction, the *shafts*, with the goal of further accelerating ray traversal. With the integration within a spatial base structure, it is guaranteed that this approach is at least as fast as the base structure, but moreover, it is demonstrated that the approach is in most cases significantly faster due to the precomputed visibility. The directional information can be diverse and depends on the actual use case. Different options are presented in this work, which lead to a variety of rendering effects that are further accelerated with this method. However, additional integration also results in the mentioned downsides. Obviously, it has higher memory consumption and initialization time, which are dependent on the granularity of the precomputations. It is therefore necessary to find a reasonable granularity, i.e. feasible parameter sets of the data structure's subdivision. Furthermore, the actual scene geometry is in some cases approximated, which may lead to visible errors depending on the use case. To limit and reduce these errors, different techniques such as integration to two-level hierarchies and a usage heuristic are proposed and evaluated.

In that sense, this work represents an important step in the final goal of the complete visibility precomputation. While it is obviously not yet possible to precompute the total visibility of a non-trivial and potentially interactive scene for every point and every direction, the presented techniques

show the current possibilities and limitations in this kind of directional precomputation.

In particular, this work explores the possibilities and limitations of the combined approach with precomputed directional information in a spatial structure with the aim of accelerating different stages of the ray tracing process. In that, this work satisfies the following objectives:

- Presentation of a combined approach between spatial and visibility data structures (i.e. the LS) with precomputed directional visibility information that is used in order to accelerate the process of ray traversal. As such, it especially extends and improves traditionally used spatial base data structures.
- Description of an intermediate abstraction (i.e. LS shafts) for constricted visibility precomputation, in order to reduce the multidimensional directional subspace and therefore limit the amount of needed memory to a feasible extent.
- Portrayal of useful types of precomputed visibility information, that can be used in order to accelerate tracing of all or specific kinds of rays (i.e. rays for shadows and indirect illumination) and evaluation of their benefits and disadvantages.
- Usage and improvement of state-of-the-art base data structures (i.e. regular recursive grids and common structures based on bounding boxes such as bounding volume hierarchies).
- Reduction of further constraints (i.e. memory consumption, initialization time and approximation errors) while improving its acceleration capabilities in the tracing of rays.

1.2 Contributions and Publications

Based on these objectives, the work at hand presents the next step in visibility precomputation for reduction of ray traversal cost. This is done through the LS integration into existing spatial data structures in order to accelerate the process of ray traversal within different rendering steps and was mostly already published at multiple peer-reviewed conferences. In the following, the main contributions are summarized and presented in a common context, as illustrated in Figure 1.1.

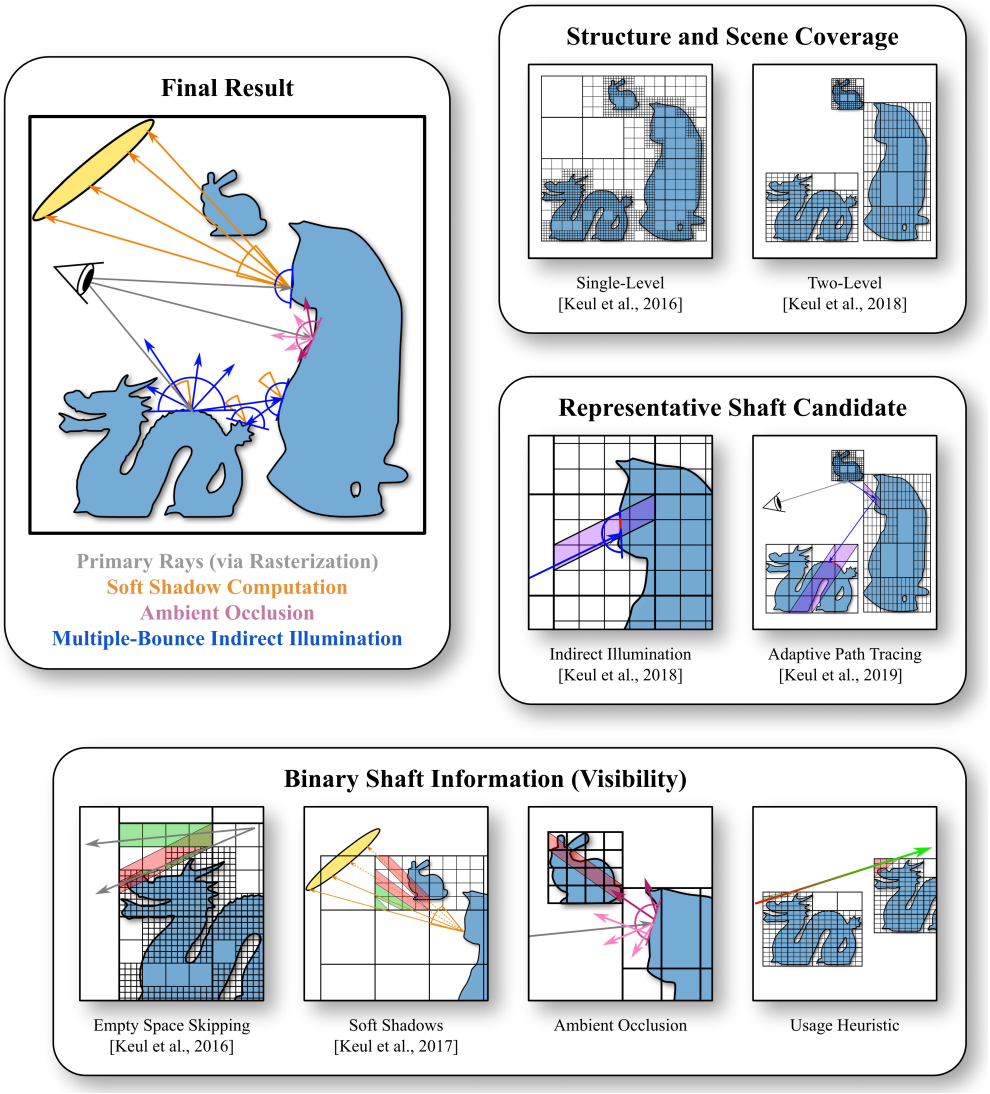


Figure 1.1: The major contributions of this work. The Line Space is proposed as an efficient directional acceleration structure for ray tracing computations based on visibility precomputation. In terms of general structure, the integration into recursive grids and an adaption to structures based on bounding boxes are presented and analyzed. Multiple techniques and improvements for ray tracing acceleration are proposed and evaluated. These include an empty space skipping procedure for all rays, an inaccurate but fast method for soft shadow and ambient occlusion calculation and an approximation-based technique for indirect lighting computation. These can be further enhanced by dynamic adaption in multiple-bounce path tracing and by incorporation of a usage heuristic. The distinction of the main contributions into subgroups and their scientific publications are shown.

- **Structure of the Line Space:** This work gives a clarification of the LS structure and its organization. The basic concept of information abstraction within shafts is explained and possible benefits and limitations are discussed. For this, the shaft composition and generation as well as typical properties of the LS are illustrated. It is further presented how the LS is used within bounding boxes and how it is integrated into typical spatial structures, such as recursive grids and bounding volume hierarchies. Consequently, an adaption to two-level structures is shown, which enables further benefits and improves LS applicability. Additionally, as the main concern is on runtime performance, GPU implementations for all these methods and efficient data packing schemes for different types of visibility information are presented.

- **Binary visibility information within shafts:** Novel approaches based on simple binary visibility precomputations are shown. Different techniques to decide the emptiness of a given shaft are proposed, discussed and used in order to initialize the LS with this information. These include initialization through clipping, masking and ray casting, which are explained and compared. A variety of techniques based on this shaft information are presented and evaluated:

  - **An Empty Space Skipping technique,** where the binary visibility information is used in order to decide, whether further inspection of the ray can be skipped. It is shown, how this is used for tracing acceleration of all types of rays and that a speed-up of up to $1.4\times$ compared to the pure base data structure is achieved.
  - **A Shadow Computation technique,** where the binary visibility information is used as early ray termination criterion in approximated shadow generation. The incorporation of LS visibility tests into shadow traversal along with the benefit of fast but inaccurate shadow generation is explained. The usage of this algorithm for soft shadow computation is depicted and therein it is shown that the loss in image quality due to approximations is quite low. Consequently, a performance gain of up to $3\times$ compared to the pure base data structure is presented.
  - **An Ambient Occlusion Computation technique,** where a disc sampling is used in order to reduce LS artifacts that are created by the above mentioned approximated shadow ray traversal. It is shown that the shadow artifacts that are created with this

technique are nearly non-visible due to the distribution of rays in ambient occlusion calculations. Hence, it is presented that the previously mentioned gain in performance can be used with nearly no loss in quality.

  - **A Usage Heuristic,** which can be used in combination with the previous soft shadow and ambient occlusion techniques. The main observations and principles of this heuristic and its application in binary occlusion determination are explained. Based on this, an adaptive algorithm that dynamically combines correct results of the base data structure with fast LS approximations is presented. As a result, a performance gain of up to 2.5× compared to the typical data structure with nearly no loss in image quality is demonstrated.

- **Representative candidate information within shafts:** Directional precomputation of shaft candidates is proposed as a novel method for ray traversal acceleration. Different techniques to compute the representative shaft candidate based on ray tracing during initialization are proposed. It is further shown how this representative candidate is efficiently stored within the data structure and how it is used to accelerate the computation of a variety of rendering effects through ray traversal acceleration and candidate approximation. The two main techniques are:

  - **An Indirect Illumination approximation technique,** where the stored candidate per shaft is used as representative for all possible rays that pass the shaft. It is explained, how the representative candidate can be extrapolated and used even for rays passing the shaft but do not intersect the candidate. Moreover, a simple method for testing the quality of the candidate approximation is described. The new algorithm's benefit in terms of performance gain is shown, which is, similar to the shadow computation technique, multiple times higher compared to the base data structure without LS usage. The drawback of candidate approximation and the resulting image error is explained and presented in detail.
  - **An Adaptive Path Tracing technique,** where the representative candidate is stored in different depths of the LS hierarchy on object level and only the needed depth is used during runtime. It is explained how the LS can be used in a two-level data structure and it is shown how this can be used to significantly reduce

approximation errors while also maintaining an improvement in runtime performance. Furthermore, an adaptive method for path tracing acceleration is proposed where high-quality approximations are used when needed and low-quality approximations with high tracing performance are used when sufficient.

In short, the following contributions are already covered in the presented previous publications and are used in this thesis:

1. LS introduction based on binary visibility information within recursive grids for ray traversal acceleration through empty space skipping with fast initialization by the use of masking. Results are presented in chapter 4 and already published in:

   Keul, K., Müller, S., and Lemke, P. (2016). Accelerating spatial data structures in ray tracing through precomputed line space visibility. In *Proceedings International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG)*, volume 24, pages 17–25

2. Approximative approach based on binary visibility information for the acceleration of soft shadow computation with GPU implementation and efficient storage by the use of data pools. Results are presented in chapter 5 and already published in:

   Keul, K., Klee, N., and Müller, S. (2017). Soft shadow computation using precomputed line space visibility information. *Journal of International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG)*, 25:97–106

3. Approximative approach based on representative candidate precomputation per shaft on the GPU for the acceleration of indirect illumination computation with LS adaption to bounding boxes and therefore all typical spatial data structures. Results are presented in chapter 6 and already published in:

   Keul, K., Koß, T., and Müller, S. (2018). Fast indirect lighting approximations using the representative candidate line space. *Journal of International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG)*, 26:11–20

4. Adaptive approach based on representative candidate approximation on object level of a two-level structure for acceleration of different

levels of a GPU path tracing system. Results are presented in chapter 7 and already published in:

> Keul, K., Koß, T., Schröder, F. L., and Müller, S. (2019). Combining two-level data structures and line space precomputations to accelerate indirect illumination. In *Computer Vision, Imaging and Computer Graphics Theory and Applications*

1.3 Methodology

To measure the benefits and limitations of the presented techniques in later chapters, different quantitative and qualitative results are gathered and discussed. The general test preparation and setup is in most cases similar and therefore shortly illustrated in this subchapter.

The main goal of the discussed data structure is, in general, the acceleration of ray traversal. For this purpose different typically used test scenes are involved with varying number of scene primitives and different characteristics. Generally, the types of scenes are dividable in object scenes that contain single mesh objects, architectural scenes as used in computer games and mixed scenes that rely on instancing of objects within an architectural scene. The used hardware is in all cases among the state-of-the-art and high-end in order to compute reliable performance values. The results are compared to the pure spatial base data structure, which demonstrates the benefits of the presented approach.

If not stated otherwise, different camera angles are used for each scene and the average median runtime over multiple rendering cycles is used as a performance value. This is presented in frames-per-second (FPS) and thereby higher values comprehensibly show better results in terms of runtime performance, which makes the comparison of the LS to the state-of-the-art competitor easier. To be more precise with the tracing performance of rays in specific rendering conditions, the values are in some cases formulated as rays-per-second. The main limitation of the approach is oftentimes its memory size as well as its construction time. These are generally measured in megabytes (MB) and seconds (s), only counting the overhead produced by the LS. To demonstrate the relation between the benefit and the limitations, bar charts for these values are shown. Typically, the memory size and the initialization time are in proportion to each other, therefore only one of those (i.e. the memory size) is presented with a bar chart. In general, the charts have a linear scaling, only in special cases an exponential scaling is used for demonstration.

Most of the acceleration achieved by the LS is based on approximations in the precomputed visibility information. This typically results in image errors, which are presented in different ways. To present the pure amount of image errors, an error value is computed. This is based on the root-mean-squared-error (RMSE) of all image pixels compared to a ground-truth image generated by the error-less competitor. Obviously, higher values of this error amount represent images with lower quality. While this value is summarized over the complete image, also per-pixel error images are presented with heatmaps. These are generally visualized in one corner of the erroneous result images and show which scene areas contain more errors. The heatmap pixel values are dark blue in areas with less errors and bright yellow in areas with high error values. The color scaling in relation to the error is linear in terms of brightness.

1.4 Outline

The goal of this work is to present the LS as a viable data structure that contains precomputed visibility information that can be accessed with nearly no traversal cost. It therefore combines directional with typically used spatial data structures in order to enhance their runtime performance in different parts of rendering. To present this, this work is structured as follows.

First, the fundamentals of this work have to be explained and set into perspective. The related work is presented in chapter 2. This includes ray tracing in general, rendering effects such as soft shadow and indirect illumination computation and path tracing, as presented in section 2.1. After this, the state-of-the-art in spatial acceleration structures is presented in section 2.3 and in terms of directional data structures presented in section 2.4. The basic principles and concepts of the LS are covered in chapter 3, which includes the relation to previous work and the general abstraction of visibility information within shafts.

Following this, the main part of this work includes the LS usage within different data structures in order to accelerate various parts of rendering. The introductory part is the acceleration of all rays through empty space skipping, which is explained and evaluated in chapter 4. This includes the basic concepts of binary visibility information as well as integration into recursive grids and mask-based initialization. Based on these results, the binary visibility information of the shafts is ported to the GPU and used for fast approximation of soft shadows, which is covered and evaluated in chapter 5. Within these two chapters, the benefit of the LS with binary

information in terms of tracing performance is shown for the general case as well as the case of shadow rays. The usage of non-binary visibility information is presented with the representative candidate approximation for indirect illumination acceleration, as explained and discussed in chapter 6. There, it is also described how the LS can be used with bounding boxes and therefore with all typical acceleration structures and how the data structure can be efficiently organized on the GPU. Consequently, the presented results show the acceleration of indirect illumination rays and the approximation errors due to the LS. Hence, an improvement to this is presented in chapter 7 by using the LS only in the object level of two-level structures. In this chapter, it is also shown and evaluated how this can be used in a path tracing system in order to enhance further acceleration based on the LS approximations while also reducing image errors. These two-level structures are then applied to general occlusion determination for soft shadows and ambient occlusion computation in chapter 8. There, this idea is further enhanced by the incorporation of a usage heuristic, which enables dynamic distinction between correct and fast but approximated results based on the object's bounding box size and distance. This heuristic is able to reduce nearly all approximation artifacts in shadow and ambient occlusion calculation while at the same time maintaining a good amount of the gain in performance.

Finally, all results are summarized and concluded in chapter 9. Additionally, future work to the work at hand is motivated and outlined.

CHAPTER 2

BACKGROUND AND RELATED WORK

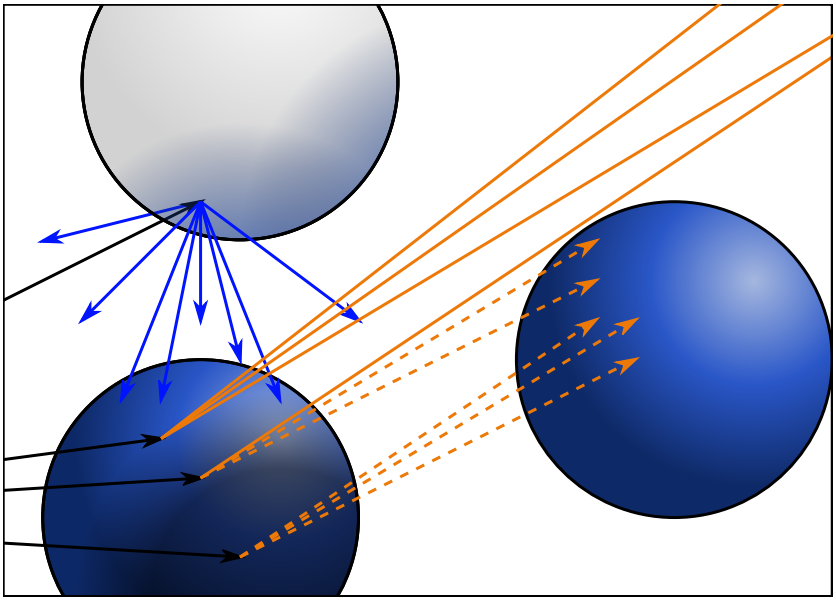


Figure 2.1: Rendering images with good quality requires the calculation of difficult lighting conditions, like soft shadows (orange) and indirect illumination (blue). Ray tracing is a way to accomplish this but needs suitable data structures to achieve acceptable performance.

The LS is an approach for combining directional visibility information within spatial data structures. As such, it is applicable in different fields of ray and path tracing systems and rendering effects. This chapter therefore presents an overview of the background that is needed for understanding the whole context.

The main topic of this work is ray tracing acceleration through dedicated data structures. First, the main fundamentals of ray tracing are explained. It starts by presenting depth-based rasterization rendering and with this the generation of simple local lighting effects. Then, ray tracing is introduced as a technique to calculate complex rendering effects, such as fuzzy reflections, soft shadows and indirect illumination. Lastly, path tracing techniques are addressed as methods for calculating global image effects resulting in photo-realistic rendering quality.

In a second part of this chapter, the focus is on data structures for ray tracing acceleration. This starts with spatial data structures, as those are the most comprehensible and by far the most used data structures for this task. Therefore, an overview of uniform grids, octrees, recursive grids, KD-trees and finally bounding volume hierarchies (BVHs) is presented. Apart from those typically used structures, the field of visibility and directional data structures is explored. These were for example used in radiosity systems, but since then their usage is declining.

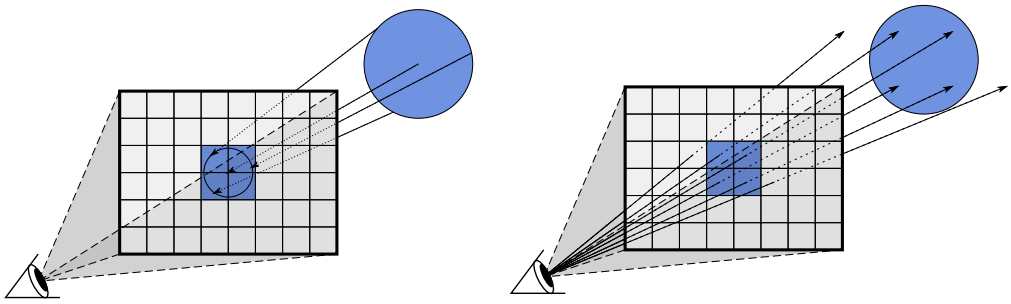
The final goal of this work is to present how these techniques can be combined by the LS and to evaluate the usefulness of those combinations in ray and path tracing systems.

2.1 Rendering Techniques

The fields of computer graphics and rendering effects span over several decades and thousands of relevant publications. For this reason, it is impossible to introduce the full spectrum of approaches, applications, techniques and scientific achievements. Therefore, only a small fraction of the most significant literature is introduced here. For further information the reader is referred to the presented literature.

2.1.1 Rasterization and simple Rendering Effects

There are two major attempts in rendering images. The first is a depth-based rasterization attempt, where all primitives of the visual scene, e.g. triangles, quads or spheres, are handled consecutively. For every such



Rasterization Ray Tracing

Figure 2.2: Schematic differences in rendering between rasterization-based and ray tracing attempts. While in the former one scene primitives are projected on the screen pixels, the latter one constructs a ray per pixel and calculates the first ray-scene intersection point.

primitive, the vertex positions are transformed to screen coordinates, indicating whether the primitive is visible at all and if so, specifying the visual primitive's total screen size. All affected screen pixels are directly identified by this process, which results in good performance. However, the color calculation of one screen pixel only has information of the current primitive and the interpolated vertex attributes. This information is sufficient for local lighting calculation, but complex scenarios like soft shadows, indirect lighting and ambient occlusion require further, global information of the complete scene. Most effects in real-time graphics are only approximated and do not result in photo-realistic image quality. For an overview of state-of-the-art techniques in real time rendering, it is referred to [Akenine-Möller et al., 2008] and [Ritschel et al., 2012].

2.1.2 Ray Tracing and complex Rendering Effects

The second attempt is done by tracing of rays, which is much more suited for good image quality. While rasterization-based attempts consecutively project scene primitives on the screen pixels, ray tracing attempts calculate the connection between scene primitives and screen pixels in a different way. There, a ray is constructed for every pixel, starting from the camera and leading through the screen at the given pixel coordinates. This ray is transformed into the scene coordinate system and traced along the scene in order to find the first intersection point with the primitives. In contrast to the fast rasterization-based primitive projection, this process is more difficult, as it represents the task of finding the first scene primitive for

Algorithm 1 Comparison of rasterization-based rendering and ray tracing.

Rasterization

```
1: for all triangle t ∈ scene do
2:     pixels ← rasterize(t)
3:     for all pixel p ∈ pixels do
4:         CalcColor(p)
5:     end for
6: end for
```

Ray Tracing

```
1: for all pixel p ∈ screen do
2:     ray ← CalcRay(p)
3:     for all relevant triangle t do
4:         i ← Nearest(ray,t)
5:     end for
6:     CalcColor(ray,i)
7: end for
```

every generated ray. For this task acceleration data structures are used, as described in section 2.2. Because of this, rasterization-based rendering has in general significantly better performance. The difference between these approaches is shown in Figure 2.2 and Algorithm 1.

However, the main strength of ray tracing attempts is not the rendering performance but image quality. In rasterization-based attempts complex lighting situations and advanced rendering effects are integrated through exploitation of per-pixel depth information in temporarily constructed images. Shadow Maps are generated from light source perspective and grant the nearest distances from this point light source to scene objects, and thus this information is used to decide, whether an object is lighted or occluded. The incorporation of area lights and soft shadows with this concept is non-trivial and results are insufficient in photo-realistic quality. The same applies also to effects like indirect lighting, fuzzy reflections, ambient occlusion and finally complete global illumination. An overview of those techniques and further information is given in [Ritschel et al., 2012].

Ray tracing systems grant further robustness and flexibility in terms of image quality. The nearest objects to a specific point are not found through projection to a virtual camera plane but via tracing of temporarily constructed rays along the scene, starting in the current camera center and leading through the examined point. This way of finding the nearest scene primitive is more difficult, as shown above, but is easily applied to other scenarios. In that sense, the decision whether a point is lighted by a light source or occluded by another object is easily decidable by tracing an additional ray, a shadow ray, starting at the object and leading to the light source. Consequently, area lights and soft shadows are computed by using multiple shadow rays to different points on the light surface. Indirect illumination and fuzzy reflections are calculated by using multiple rays starting in the object but leading in different specified directions and collecting the lighting state in the newly found intersection point. This

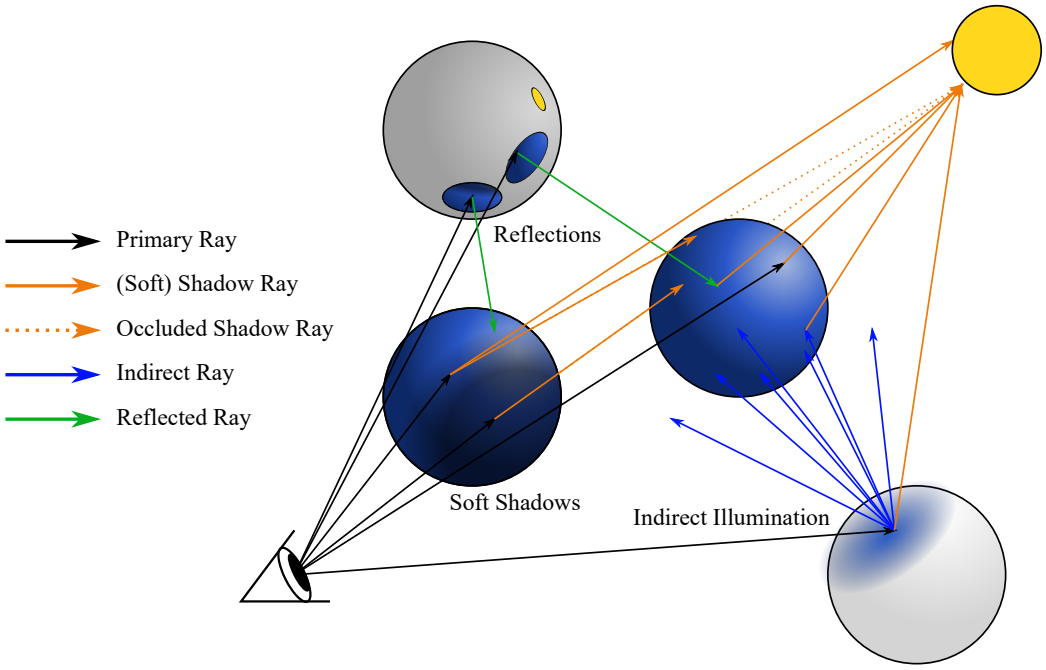


Figure 2.3: Schematic illustration of different effects calculated via ray tracing. Primary rays (in black) start in the camera and find the nearest scene object. From there on further rays are used to calculate simple and soft shadows (in orange), reflections (in green) and indirect illumination (in blue).

concept is shown in Figure 2.3. Further information on ray tracing is presented in [Glassner, 1989].

These two ways to render synthetic images, rasterization and ray tracing, have contrary strengths and weaknesses and therefore present different trade-offs between performance and realism. Rendering through rasterization is typically a lot faster compared to ray tracing systems and, in conjunction with dedicated hardware, even capable of rendering images in real-time, which is crucial in terms of computer graphics in games. Ray tracing is, depending on the demands in realism, incapable of real-time rendering. However, as stated above, results can achieve realistic appearance, which makes it a suitable process for rendering films and movies. The focus of this work is on acceleration of ray tracing systems through suitable data structures. The following background and literature references concentrate on these topics.

The main challenge in ray tracing is the determination of the nearest intersecting scene primitive. This task in general scales with the number

of scene primitives as well as the screen resolution. Increasing the performance of this task is therefore crucial, and there exist different attempts, as explained in the following paragraphs. In the scope of this work, the main acceleration part is in detecting the most suitable data structure for finding the nearest intersection point. More background on acceleration data structures is presented in [Havran, 2000] as well as section 2.2.

Using dedicated hardware is a significant topic in computer graphics, exploiting GPU (graphics processing unit) power is a prominent way in accelerating algorithms. In terms of ray tracing, one such way is to make heavy use of SIMD (single instruction, multiple data) commands and by this taking advantage of the GPU's full computing capability. In this context it is important to rely on coherency in data and ray management. An overview on this is presented in [Wald et al., 2001]. Fast results in ray tracing are especially required in hybrid techniques, which combine rasterization and ray tracing in order to extend real-time graphics with simple convincing effects, such as reflections. Consequently, current progress tends to use ray tracing in interactive scenes. More information is presented in [Wald et al., 2009]. Recently, a new generation of dedicated graphics hardware with incorporated ray tracing components was introduced by NVidia with the RTX-series [1]. There, special ray tracing cores are used for hardware-accelerated construction and tracing of data structures. For using ray tracing in applications, there exist a vast diversity of different frameworks. Two of them are NVidia Optix [2] [Parker et al., 2010] and Intel Embree [3] [Wald et al., 2014]. They already incorporate most of the developed acceleration techniques and are used in a significant amount of applications.

Apart from enhancing the performance of intersection finding, increasing the efficiency of ray shooting and therefore decreasing the number of needed rays is a prominent way to boost the overall performance of ray tracing. For realistic image generation different subsequent rays, starting in the camera and finally leading to a light source, are grouped to paths. The usage of paths for photo-realistic rendering is called *path tracing*. Then, the main goal is to reduce the total number of generated paths by only using those paths that contain significant information of the examined point's lighting situation. This kind of acceleration is further explained in the following subsection.

[1] NVidia RTX: `https://www.nvidia.com/de-de/geforce/20-series/rtx/`
[2] NVidia Optix: `https://developer.nvidia.com/optix`
[3] Intel Embree: `https://embree.github.io/`

2.1.3 Path Tracing and Photo-Realistic Effects

The most complex lighting situations arise from light that is scattered and reflected multiple times within the scene before it arrives in the viewer's position. This is known as global illumination, opposed to local lighting, and requires the examination of a significant amount of lighting particles such as photons. Those are irradiated by light sources and the subsequent analysis of their paths along the scene is known as path tracing. Every single segment of these paths can be calculated via ray tracing and for each intersection found, the next direction is calculated based on the material properties in the found intersection. By that, the physically correct distribution of light is simulated within the scene. A first formal description was presented with the *rendering equation* by [Kajiya, 1986] as follows:

$$\begin{aligned} L_o(p, d\vec{\omega}_o) =& L_e(p, d\vec{\omega}_o) + \\ & \int_{2\pi} f_r(p, d\vec{\omega}_i, d\vec{\omega}_o) * L_i(p, d\vec{\omega}_i) * cos\theta_i d\vec{\omega}_i \end{aligned} \tag{2.1}$$

This results in the outgoing radiance L_o in direction $d\vec{\omega}_o$ from surface point p of a scene object. It is computed by two components. The first is the self-emitting radiance L_e of the surface point in the presented direction. The second is the sum of all reflected light L_i from all possible incoming directions $d\vec{\omega}_i$, additionally weighted by a material function f_r and the incoming lighting angle θ_i. The material is described by a *bidirectional reflectance distribution function* (BRDF), which in general computes how much light is reflected from incoming direction $d\vec{\omega}_i$ to outgoing direction $d\vec{\omega}_o$ of surface material at point p. These directions in the BRDF are defined by their azimuth angle ϕ and zenith angle θ, which makes the BRDF in the simple form a 4-dimensional function. Further attributes, like the material's refraction, increase the dimensionality of this function. For more information of BRDFs and physically correct rendering in terms of material, it is referred to [Pharr et al., 2016].

A reformulation of the rendering equation was presented by [Veach, 1998] with the *three-point form*. There the directional approach is replaced by the light transport between surface points:

$$\begin{aligned} L(x' \to x'') =& L_e(x' \to x'') + \\ & \int_S L(x \to x') f_s(x \to x' \to x'') g(x \leftrightarrow x') dx \end{aligned} \tag{2.2}$$

Again, the result is the transferred radiance L, but this time between points directly. The radiance between points x' and x'' is computed by two components, similar to before. The first is the emitted light of point x' to x''.

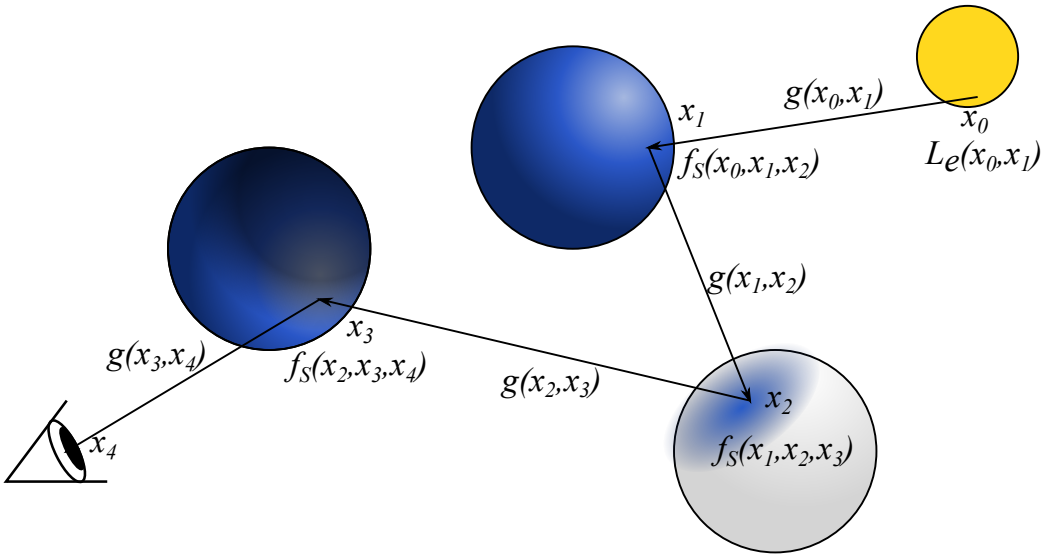


Figure 2.4: Example of a single path starting from the light source (with emitted radiance L_e) and ending in the eye. This path consists of 4 single rays, each connecting to different intersection points x. The function f_S describes the BRDF at the given intersection point and therefore the sampling function used for the next ray direction. The g-function represents the geometric term, which is used for general visibility.

The second is the sum of all incoming radiances by all other surface points x of scene S towards surface point x'. Those are weighted by the BRDF f_s, as before (here surface points instead of directions are used). Additionally, a geometric factor g between points x and x' is necessary, which states how much of x' is visible from x. The three-point form is similar to the original directional description but in the case of some path tracing variants more useful.

The main difficulty of both variants is the integral's correct computation, which accumulates the light from all directions. This integral is in general impossible to solve and therefore Monte-Carlo methods are used to estimate it's value [Dutré et al., 1993]. For this estimation, samples are used that start at a surface point of a light source with a given emitted radiance quantity. At every intersected surface of the scene objects, a new direction is calculated according to a random sample of the BRDF. This process ends when the sample reaches the viewing plane. It is then evaluated in terms of the remaining radiance value and thus changes the color of a certain pixel of the image plane. This is shown in Figure 2.4. As with typical Monte-Carlo integral estimation, with increasing amount of samples this procedure

converges to the correct result. However, this process produces noticeable noise with insufficient amount of samples and therefore requires significant computational amount. One way to reduce this is to start from both ends of the resulting path, from the viewing plane as well as the light surface. This process is known as *bidirectional path tracing* and is able to reduce the computation time needed to render images with difficult lighting states [Lafortune and Willems, 1993] [Veach and Guibas, 1995a]. Another way of reducing the produced noise, is by using better sampling strategies for subsequent path segments. This can be achieved with different ways. One is to use multiple sampling strategies and combine those according to their calculated importance, which is known as Multiple Importance Sampling [Veach and Guibas, 1995b]. Lastly, the number of necessary rays can be reduced by using techniques that offer the possibility to generate new paths with high importance by divergence of already calculated important paths. This is summarized in the Metropolis algorithm [Veach, 1998], where multiple methods for promising path divergence are introduced.

A different approach on rendering photo-realistic images is by using photon mapping [Jensen, 1996] [Hachisuka et al., 2008]. There, photons are traced during initialization time. They start at the light source and are stored within a dedicated data structure, usually a binary space partitioning tree. During runtime, the stored photons around a specific point are evaluated, which leads to a physically correct lighting of difficult effects like caustics. This process was later also accelerated through the GPU [Davidovič et al., 2014]. Depending on the generated lighting effect photon mapping leads to significantly less or more noise compared to path tracing systems. Because of this, newer approaches try to combine these techniques to benefit from both of their advantages in reducing noise [Georgiev et al., 2011] [Georgiev et al., 2012].

All these approaches try to reduce the number of rays that need to be processed for good image quality in order to increase overall performance. A different method for this goal is to increase the rays' total tracing performance. This is achieved by using suitable acceleration data structures, as explained in the next section.

2.2 Acceleration Data Structures

In all ray tracing systems, the main cost of runtime performance is the tracing of rays. For every ray, the first intersection with the scene objects needs to be found. In a naive approach all scene primitives need to be processed in order to find this intersection point. Thus, the problem can be

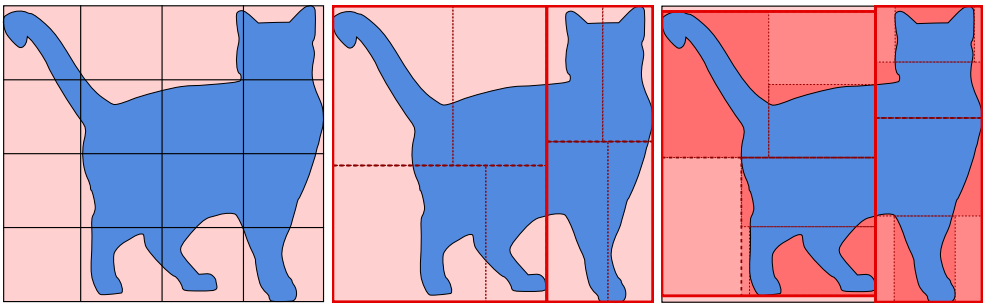
considered as search problem, which can be accelerated through the use of suitable data structures.

In the context of this work, the presented data structures are distinguished in two categories: spatial and directional structures. The former are typically used in this scenario and represent all structures that subdivide the scene space according to its spatial size and the distribution of it's contents. This category consists of grids with fixed and dynamic subdivision schemes, tree structures with grid-like behavior, binary space partitioning trees and bounding volume hierarchies. The second category consists mostly of visibility data structures that were used in scenarios like radiosity computations [Cohen and Wallace, 1993]. The structure proposed in this work, the LS, is a combined approach between these topics.

2.3 Spatial Structures

Structures building around a spatial subdivision of scene primitives have been studied extensively. Therefore this section only presents a brief presentation of the most relevant literature. For a general introduction in ray tracing and acceleration techniques it is referred to [Glassner, 1989]. In the context of spatial acceleration structures, there have been quite a few comparative studies, showing the strengths and weaknesses of different structures. [Havran, 2000] is one of the most relevant of these studies. More specifically, various comparative studies on GPU algorithms were presented, one of which is [Zlatuška and Havran, 2010]. In those the two main and commonly shown structures are KD-trees and bounding volume hierarchies (BVHs). Because of their relevance and the broad field of improvements different studies specifically focus on these two structures, such as [Vinkler et al., 2016]. A typical observation is that the BVH is the most used data structure, which is due to it's versatility and the vast field of improvements and hierarchy-optimizations of recent research. For further enhancements in tracing performance, these structures have been used in conjunction with specifically designed hardware. In the context of this work, acceleration through GPU involvement is focused and therefore in this section particularly covered. Other approaches like field programmable gate arrays (FPGAs) [Schmittler et al., 2004] [Woop et al., 2005] or similar projects, such as the cell processor [Benthin et al., 2006], are not used in this work and hence out of the scope.

This section therefore presents an overview of different structures. Some of these are presented in Figure 2.5. These are:



Uniform Grid KD-Tree BVH

Figure 2.5: Some of the typical subdivision structures. **Left:** Grid structure with uniformly sized cells. **Middle:** KD-tree with binary space subdivision. **Right:** BVH with bounding volumes that further narrow the space around the object.

1. **Grids**, which have a specific subdivision parameter per dimension. In this context also recursive grids are mentioned, as those are the first base structure for the LS.

2. **Octrees**, which basically are a special case of recursive grids with a subdivision parameter of 2 per dimension. Also sparse voxel octrees (SVOs) are mentioned, as they are in special cases used for acceleration of shadow computations.

3. **KD-trees**, which along with BVHs are the most typical and commonly used structure for ray tracing acceleration. Although their popularity declined compared to the BVH, many of the improvements and statistical measurements of BVHs were first developed for KD-trees.

4. **BVHs**, which are the main competitor of the here proposed structure. Moreover, they are later used as the main base structure for the LS.

2.3.1 Grids

The simplest spatial subdivision scheme is to order objects within equally-sized cells in a grid spanning the whole scene. In terms of ray tracing, two main tasks arise from this: the initialization of a grid structure with the according arrangement of scene primitives in the grid cells and the ray traversal of this grid structure during runtime. This provides a significant boost in ray tracing performance, as no longer all scene primitives need to be tested for intersection with a ray, but only that small fraction of the total amount that is arranged in the ray's traversed cells. In that,

the total tracing cost can be divided in traversal of the underlying data structure and intersection cost of the remaining scene primitives. Most traversal algorithms for this data structure are based on a 2D line drawing algorithm [Bresenham, 1965], such as the 3D-DDA (3D Digital Differential Analyzer) algorithm [Amanatides et al., 1987]. More advanced algorithms use coherent grid traversal [Wald et al., 2006] and are able to efficiently use parallel GPU computations.

The grid cells are also known as *voxels* (from volume elements) and the initialization process therefore known as *voxelization*. Many approaches were proposed in order to generate such grids with high performance. The main concern of grid subdivisions is to generate those in real-time, for example through dedicated hardware usage [Fang et al., 2000] and utilization of GPU rendering with efficient encoding of the voxels [Dong et al., 2004]. Therein, triangles as scene primitives are rendered from each of the grid's main sides in order to identify the occupied voxels. The high parallel performance of the GPU was utilized to concurrently determine those voxels for each scene primitive. A significant improvement was achieved with GPU-accelerated voxelization algorithms, where the grid side with the highest projected primitive surface was determined on-the-fly [Eisemann and Décoret, 2006]. While first approaches use multiple GPU passes for this process, also single-pass voxelization was proposed [Eisemann and Décoret, 2008], which increases the initialization performance. With the rise of GPUs for general purpose computation, the generation process of grids was further enhanced by reducing it to a sorting problem of primitives and cell coordinates [Kalojanov and Slusallek, 2009]. This was extended to a parallel approach for surface and solid voxelization [Schwarz and Seidel, 2010] and sparse voxelization techniques based on octrees [Crassin and Green, 2012].

Applications of grid subdivision in the context of ray tracing therefore mainly focus on initialization in real-time. There, the usage of animated scenes is enabled by a per frame reconstruction of the data structure [Wald et al., 2006]. A similar technique was also used for global illumination based on voxels [Thiedemann et al., 2011]. A different approach for rigid object dynamics is to use two-level data structures, where a grid is generated for every object along with an additional structure that gathers those object structures [Kalojanov et al., 2011]. Grids were also used in conjunction with objects that do not fit into GPU memory, leading to out-of-core rendering [Gobbetti and Marton, 2005] or to render volumetric data sets [Crassin et al., 2009]. For further information on the usefulness of grids it is referred to [Hapala et al., 2011b].

While the initialization can be done in real-time, the applications of grids for ray tracing have a significant downside. Because of the strict subdivision scheme all cells have the same size, which leads to empty cells well as cells containing many scene objects at once. This is also known as teapot-in-a-stadium problem. For better scene coverage, irregular subdivision schemes can be used for this purpose [Pérard-Gayot et al., 2017]. Also adaptive subdivision schemes were proposed [Jevans and Wyvill, 1989], where specific cells can be subdivided as well, leading to recursive grids with a tree structure. Other approaches of hierarchical data structures focusing on good scene coverage are presented in the following.

2.3.2 Octrees

While recursive grids have an arbitrary subdivision parameter for each dimension, octrees (or in 2D quadtrees) have the smallest possible parameter, opting for reduced needed memory. Every node is therefore subdivided in 8 (=2*2*2) equally sized subnodes, which are subdivided as well until a specific criterion (like the maximum depth) is reached. Octrees for ray tracing were proposed in a bottom-up manner through the use of neighbor finding [Samet, 1989]. An improvement in the general traversal strategy was proposed [Revelles et al., 2000], where it was done top-down instead of bottom-up. The construction of such trees was improved by using GPU accelerated algorithms. One example is to first use an intermediate binary radix tree that consists of binary space partitioned areas with their specifying indices ordered in parallel within the tree. Octrees and other hierarchical data structures are then constructed based on this intermediate tree, as presented in [Karras, 2012].

Further advancements were achieved with sparse voxel octrees (SVOs) that had lower memory consumption by not including those subnodes that only contain empty space. Their generation is usually done based on already voxelized scenes, where the voxels are then combined to trees. A special contour replacement of the voxels allowed for an efficient and fast ray tracing [Laine and Karras, 2011]. In terms of real-time rendering SVOs were used for voxel cone tracing [Crassin et al., 2011], which was also able to handle animations by using dynamic updates within the voxel octree data structure. By identifying identical regions in a scene and re-using the generated subtrees, the SVO can be transformed to a directed acyclic graph (DAG), which in turn significantly reduces the needed memory [Kämpe et al., 2013]. This was especially used in shadow computation, which also could be precomputed and stored within the DAG

[Sintorn et al., 2014]. In all those applications, octrees benefit from their hierarchical structure and therefore the elimination of empty space stored in the data structure. However, they rely on equally-sized subnodes, which is impractical in specific scenes, as summarized in the teapot-in-a-stadium problem. In the following, hierarchical data structures with variable subdivisions are presented.

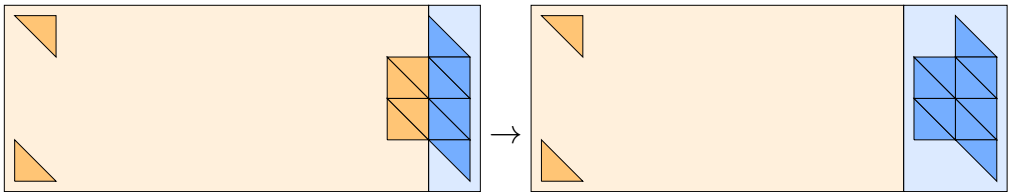
2.3.3 BSP trees and KD-trees

KD-trees and binary space partitioning (BSP) trees are among the most logical spatial data structures in terms of binary subdivision. Usually, the goal is to reduce the remaining primitives by a plane subdividing them in half and thereby granting an intersection search with logarithmic complexity. Though, this comes with some problems as most scene primitives in dense areas are not trivially dividable by a simple plane subdivision. Therefore, the most common solution is to arrange those in both subnodes, which reduces the data structure's potential performance. Because of this, one of the main tasks is to find the most suitable splitting plane in order to improve the structure's potential.

General BSP trees involve median splitting planes without further constraints [Sung and Shirley, 1992]. This median split is the simple approach of dividing scene primitives in half, but the result is not desirable when scenes containing dense areas are involved. In those cases, it is more rewarding for ray tracing applications to additionally consider the probability of the ray intersecting this area. When many primitives are centered in a specific area and few are positioned far away from this center, then the median split returns two subnodes which contain the same number of primitives. However, additionally involving the ray's intersection probability results in different splits, as demonstrated in Figure 2.6. This is used in the surface area heuristic (SAH), originally introduced for BVHs [Goldsmith and Salmon, 1987]:

$$\begin{aligned} C_S &= C_t + C_i \left(\frac{SA(V_L)}{SA(V_N)} * \#T_L + \frac{SA(V_R)}{SA(V_N)} * \#T_R \right) \\ C_N &= C_i * \#T_N \end{aligned} \tag{2.3}$$

There, it is determined whether the cost of a split C_S is lower than the current node's own cost C_N. The latter is determined by the number of primitives that are contained within the current node and therefore need to be traced, given by $\#T_N$, along with the cost of a single intersection test C_i. The cost of a split is determined by the predicted intersection cost of the



Median Split SAH Split

Figure 2.6: Different splitting strategies for binary space partitioning. **Left:** Median splitting is used, resulting in two subnodes containing the same number of primitives. **Right:** Additionally the Likelihood of a ray intersecting the subnodes is considered. The surface area heuristic (SAH) involves the subnodes' bounding volume size for this.

subnodes in addition to the cost of one traversal step C_t. The intersection cost of each of the two subnodes (denominated by L for the left and R for the right one) is computed by their specific number of contained and tested scene primitives $\#T_L$ and $\#T_R$ in consideration of the probability that a ray intersects these subnodes. That probability can be calculated by the volumes' surface areas $SA(V_L)$ and $SA(V_R)$ in relation to the current node's surface area $SA(V_N)$.

In terms of BSP trees, the SAH was applied to result in restricted BSPs that have lower initialization time [Kammaje and Mora, 2007], however also worse tracing performance. For a better trade-off between initialization and runtime performance KD-trees are typically used, which restrict the splitting plane of the BSP to be parallel to one of the axis planes. This restriction guarantees higher build rates but obviously unrestricted BSP trees typically result in better runtime performance, as an arbitrary splitting plane can be chosen which is able to further improve tracing speed [Thiago Ize, 2008]. Nevertheless, BSP trees are almost always used in the form of KD-trees, as the better trade-off is usually preferred.

KD-trees have been heavily studied in ray tracing acceleration. Because of this, only the most significant advancements in hierarchy construction and traversal are presented here. The latter is in general mostly straightforward, as shown in [Hapala and Havran, 2011]. First KD-trees used a median split based on the number of scene primitives, which are spatially subdivided through a separating plane that is parallel to one of the axis planes [Kaplan, 1985]. As shown above, the additional consideration of the subnodes' intersection probability through the SAH improves tracing performance due to more adequate splits. The current node's splitting plane is arbitrarily chosen at every endpoint of a scene primitive, resulting

in 2#P possibilities (with #P representing the number of primitives in the node) [Wald and Havran, 2006]. For each of those the SAH is evaluated in order to decide which split is the most promising. However, due to the huge amount of possible splits, this process is quite slow. To improve the initialization speed, binned construction was proposed [Popov et al., 2006], where only a small number of equally distanced possible splits were tested. By this, the initialization performance was significantly increased, although tracing speed was reduced by a minimal amount. It was further improved through better use of memory coherency, which accelerates the construction and finally lead to dynamic creation in real-time [Shevtsov et al., 2007].

Computation on the GPU enables better performance, first by generating the data structure offline on the CPU and using the results for ray tracing on the GPU [Foley and Sugerman, 2005]. Initially, the KD-tree was implemented based on a stack using textures [Greiner, 2004]. This was improved by incorporation of more efficient GPU branching abilities [Horn et al., 2007]. The general approach of GPU traversal was improved by a stackless technique, where only a minimal reference is stored as reminder of the current tracing state [Popov et al., 2007], which was more suitable for limited GPU cache memory and resulted in better runtime performance. Adapting the construction algorithm to GPU acceleration further reduced initialization time, but due to the more difficult handling of tree structure generation on parallel GPU hardware the tracing performance was lower [Danilewski et al., 2010]. A hybrid approach between CPU and GPU sharing the main tasks improved construction and traversal further [Roccia et al., 2012].

It was observed that highly coherent rays use similar subtrees of the KD-tree and produce less memory traffic, therefore resulting in higher tracing performance in those cases. This lead to the development of packet ray tracing [Wald et al., 2003b], where instead of a single ray packets of multiple rays are traced. Obviously, this is easily achievable with primary rays, as the screen pixels are simple subdivided in groups of $n \times n$ pixels (with n for example 8). This concept of packet tracing was especially used with KD-trees [Wald, 2005], but as this is not the focus of this work, it is not handled here.

KD-trees have been among the most used data structures for ray tracing, but their usage has been declined significantly. BVHs, as strongest competitor, grant more possibilities for improvement and subsequent optimization and are recently typically used. While there exist hybrid approaches between KD-trees and BVHs [Havran et al., 2006], it was shown that the latter grant faster initialization, less memory consumption and bet-

ter tracing performance [Zlatuška and Havran, 2010] [Vinkler et al., 2014] [Vinkler et al., 2016].

2.3.4 Bounding Volume Hierarchies

Similar to KD-trees BVHs aim to subdivide the scene in a spatial manner along a promising splitting plane. The main difference is that KD-trees only store the splitting plane per node, which is then used for traversal computation, while BVHs also store the bounding volume's extend (usually as an axis-aligned bounding box, AABB) per node. This way, it is possible to calculate for each subnode separately whether it is intersected by the current ray and therefore traversal continues [Kay and Kajiya, 1986]. In turn, subnodes within BVHs may be overlapping and hence oftentimes both subnodes need to be tested even if an intersection is found within the first subnode. In general, BVHs have the same or even higher runtime performance compared to KD-trees with less memory cost and more optimization potential. In fact, two of the most popular ray tracing frameworks (NVidia Optix [Parker et al., 2010] and Intel Embree [Wald et al., 2014]) as well as the newest generation of hardware-accelerated ray tracing GPUs (NVidia RTX) are based on BVHs. The bounding interval hierarchy (BIH) [Wächter and Keller, 2006] and similar approaches [Zachmann, 2002] aim to combine the ideas of KD-trees and BVHs. They are usually structured like BVHs but contain two planes per node, indicating the extends of the included children along with an order. However, due to the combined structure it is not able to benefit from the more sophisticated advancements of either of the underlying data structures. While a lot of effort was spent to improve ray tracing with KD-trees, BVHs have been long ignored in this context and were mainly used for collision detection. However, all of the KD-tree advancements could be transferred to BVHs, such as GPU involvement [Lauterbach et al., 2006], packet tracing [Gunther et al., 2007] and binned SAH splitting [Wald, 2007], resulting in the same and even higher acceleration performance. Furthermore, the splitting of objects instead of space enables object animations and deformations in the context of ray tracing scenes [Wald et al., 2007]. Due to the big amount of research in BVHs the following part is divided in traversal, CPU construction with good tree quality and GPU construction with good initialization performance.

Traversal

Typically, BVH traversal is done based on a stack that stores information of the current traversal step. While KD-trees grant the advantage that the first intersection found is actually the foremost, BVHs don't guarantee this due to possibly overlapping subnodes. Hence, it is necessary to compute the subnode ordering as well as the current distance of the temporarily found intersection point during traversal. The most basic algorithm [Kay and Kajiya, 1986] implements the remaining subnodes within a heap, which is ordered with respect to the intersected node's distance from the ray's starting point. For shadow rays, it is not necessary to compute the actual intersecting candidate, but to only compute whether the ray is intersected. This way, shadow ray traversal can be accelerated by stopping when the first intersection point is found [Smits, 1998]. As with KD-trees, this algorithm was accelerated by the incorporation of packet tracing [Gunther et al., 2007].

This traversal was also enhanced to work on multi-core systems such as GPUs [Lauterbach et al., 2006]. In those systems it was observed that high ray coherency is a way of enhancing traversal performance. One way to achieve this is by using more than two subnodes per node and therefore better SIMD (single instruction, multiple data) instructions [Dammertz et al., 2008]. This results in multi bounding volume hierarchies (MBVHs, also called wide BVHs) [Ernst and Greiner, 2008], which allow for efficient coherent SIMD ray traversal and less memory traffic without the need of packets [Wald et al., 2008], leading to some of the fastest primary ray traversal algorithms on the GPU [Aila and Laine, 2009] [Aila et al., 2012]. For incoherent rays the performance in those systems is usually significantly lower. To counteract this, a stream tracing algorithm was introduced that uses a ray stack in order to improve performance for coherent as well as incoherent rays [Tsakok, 2009]. A similar approach was presented, where a hybrid technique automatically decides whether to use a packet or a single ray tracing scheme, depending on the rays' divergence [Benthin et al., 2012]. A dynamic ordering of rays and grouping them to packets was used as further improvement to this [Barringer and Akenine-Möller, 2014] and later extended by a more adequate distinction and handling of ray packets by a MBVH and single rays by ray streaming [Fuetterling et al., 2015]. The incoherency of rays and the resulting memory traffic can be reduced by using specific compression within MBVHs that can be maintained during traversal, as presented in [Ylitie et al., 2017]. A different tactic for more coherency, which was also applied to KD-trees, is by using a stackless approach. This was shown to

use significantly less local memory in CPU [Hapala et al., 2011a] and also GPU tracing [Áfra and Szirmay-Kalos, 2014]. It can be further improved by a more advanced technique for computation of the next node, that can be efficient in time and state memory [Binder and Keller, 2016].

Methods to differentiate BVH Construction Algorithms

Efficient and effective construction of high-quality BVH hierarchies is one of the main research topics and quite a lot of different approaches were proposed for this. Therefore, it is crucial to classify those algorithms based on different characteristics, as shown in Table 2.1.

High-Quality Construction using the CPU

Generation of a BVH on the CPU is typically done top-down, as this is the most straight-forward concept. As with KD-trees, a splitting plane needs to be found, which then separates the contents to two subnodes. For this, the SAH can be used along with binned acceleration. This is more thoroughly discussed in [Popov et al., 2009]. On systems with many integrated cores, better usage of SIMD commands along with binning and SAH evaluation was shown to result in good construction and rendering times [Wald, 2012]. In case of MBVHs, multiple splitting planes can be used. This top-down process can be accelerated by the previous identification and distinction of triangle groups to mini-trees, as proposed by the Bonsai algorithm [Ganestam et al., 2015]. All mini-trees are constructed using SAH splitting with high quality, which is also used for the top tree that unites the mini-trees. The smaller parts of the hierarchy are then pruned to achieve better total tree quality.

As before, a problem with the top-down approach is when objects or primitives are intersected by the splitting plane and one of the subnodes needs to be chosen arbitrarily. The bounding volumes of those subnodes then overlap each other and when a ray intersects both of these subnodes, both need to be traversed separately, as it is not clear which contains the nearest intersection with it's content. This problem is especially noticeable when primitives with significantly size differences are used within the scene. A way to counteract this is to arrange the affected objects and primitives into both subnodes and therefore contain two references to it within the hierarchy, as proposed with the split BVH (SBVH) [Stich et al., 2009], which was also used in conjunction to more complex BVH construction algorithms like the Bonsai algorithm [Ganestam and Doggett, 2016]. There, the subnodes' sizes are unchanged, which results in better tracing performance at the cost of higher memory consumption. Consequently, SIMD

Based on the used hardware. The most common are:	
CPU generation	with the primary goal on good tree quality or
GPU generation	with focus on real-time generation and updates
Based on their generation structure. Construction can be made:	
top-down	by recursively separating the contents of the current node along a promising split plane as it was done with KD-trees
bottom-up	by first creating the leaf nodes from triangle references and then clustering nearby nodes and grouping them to parent nodes
insertion-based	by generating and changing the structure when new primitives are constructed
Based on their applicability concerning dynamic objects in interactive scenes. This can be achieved by:	
full reconstruction	using full reconstruction of the BVH, which demands fast generation. This is mostly achieved by fast GPU builders with lower tree quality and therefore reduced tracing performance.
refitting	using identification of changed BVH subtrees and adjusting only those in a refitting step. Because of the global hierarchy information needed, this is mostly done in high quality CPU approaches that would not achieve interactive rebuild. However, a degradation of the data structure and a loss in tracing performance is expected over time.
dynamic limitation	using a limitation of the object's dynamic properties and therefore predicted movement that can be either precomputed or accomplished by a two-level data structure. This is less flexible but allows for fast simple object dynamics.

Table 2.1: Different categories used to distinguish BVH construction algorithms. While this is not done in this work, these categories can also be applied to other hierarchical acceleration structures.

parallelization capability with according memory management and load balancing results in higher initialization performance while maintaining the high tree quality produced by the SBVH algorithm [Fuetterling et al., 2016]. While all previous approaches use the SAH as method to predict the structure's performance, it was shown that other heuristics may be more beneficial for this cause [Aila et al., 2013]. Consequently, different heuristics were shown to result in equal and some cases even higher performance [Wodniok and Goesele, 2017].

Apart from top-down construction, it is possible to build BVHs in a bottom-up manner. There, every single triangle is handled as leaf node with it's own bounding volume. Nearby nodes are then recursively combined within parent nodes until all nodes are collected to the final tree. Common clustering algorithms can be used for this task, like the agglomerative clustering [Walter et al., 2008]. The results are similar in runtime performance, but had higher initialization time, which was also accelerated by a complex intermediate KD-tree. This was enhanced by a binary spatial structure and an approximate neighbor search, resulting in approximate agglomerative clustering [Gu et al., 2013] with construction and tracing performance comparable to top-down approaches.

While previously mentioned approaches construct BVHs from scratch, it is also possible to change an already constructed hierarchy. This was used for further optimizations through tree rotations [Kensler, 2008], that are found via hill climbing and simulated annealing methods for optimal SAH values. While this lead to trees with higher quality, it was expensive and later enhanced by a faster and more general approach, where subtrees are removed and reinserted according to the SAH cost model [Bittner et al., 2013]. This concept was further refined to result in an incremental construction algorithm [Bittner et al., 2015]. Concerning object dynamics in interactive scenes, most CPU construction algorithms do not achieve fast build times and therefore do not offer the possibility of full rebuild. However, the changed parts of the BVH can be adapted during runtime. It is crucial, to identify the changed subtrees and only focus on those, which can be done through detection of changed BVH clusters [Garanzha, 2008]. This was extended by more sophisticated refitting methods targeting identification of empty space and recognition of overlapping space between moving nodes of the hierarchy [Yin and Li, 2014].

Fast Construction using the GPU

Generation of BVHs on the GPU typically reduces construction time significantly and thereby renders the possibility of interactive reconstruc-

tion during runtime. This can be done by reducing the construction to a sorting problem, which can be efficiently computed on the GPU. The reduction step is done by creating a linearized version of the spatial scene, resulting in the linear BVH (LBVH), through usage of the Morton curve [Lauterbach et al., 2009]. This is a special function, which is able to assign a binary representation (the Morton code) to a point in space, which is done for all geometric scene primitives based on the barycenter of their surrounding AABB. These Morton codes are then traversed in a specific order, which is called the Morton curve, and finally arranged in a linear list representation. The list is then ordered by using an efficient radix sort. By using the Morton code of a primitive, it's spatial location is computable. Following this, clustering of the bits leads to a spatial subdivision of the scene. The foremost bit then is able to split the scene space in half, resulting in two subnodes from the root node. The second bit recursively splits those in half. This process is repeated for all Morton code bits. The resulting BVH has a bad tree quality, as the subnodes are not dependent from their content. However, the construction is significantly faster due to the reason that all subprocesses rely on heavy usage of the GPU and data-parallel computation. The LBVH was improved by a hierarchical setup, resulting in the hierarchical LBVH (HLBVH), which reduces memory bandwidth through spatial coherency [Pantaleoni and Luebke, 2010] and efficient work queues [Garanzha et al., 2011]. This can be further improved by using a binary radix tree and parent-pointers during tree generation in a bottom-up manner, which enables higher parallelization during initialization [Karras, 2012]. There, it was also shown that the Morton curve can be used for octree and KD-tree generation. By using linear ranges of keys per inner node, the parent-search and bounding box generation were combined in a single bottom-up traversal, which further improved and simplified the algorithm [Apetrei, 2014]. However, Morton codes with regular ordering per axis result in equal distributions for each dimension, which is disadvantageous for scenes with unequal extents. An extension to this was proposed, where an adaptive ordering of the axes is applied, which leads to better scene coverage in those scenes and hence increased coherency of the found clusters [Vinkler et al., 2017]. Recently, BVHs and LBVHs were constructed based on loose octrees [Gu et al., 2018], an octree variant where every node is enlarged and therefore has a more detailed coverage of the scene resulting in more balanced trees.

While all these GPU construction algorithms grant interactive per-frame reconstruction, they also lead to much less tracing performance because of lower tree quality. Consequently, a method was proposed to increase tree quality in a post-processing step on the GPU [Karras and Aila, 2013].

There, treelets are generated by using a specific number of connected subnodes of an already constructed low-quality BVH. These treelets are then restructured in parallel and their SAH costs evaluated and compared, and if a new SAH cost is lower, this restructuring is used for the BVH. By using a greedy approach, this method was enhanced to use larger treelet sizes resulting in higher optimizations [Domingues and Pedrini, 2015]. More recently, the sequential insertion-based algorithm for optimization [Bittner et al., 2013] was enhanced to work in parallel on a GPU structure [Meister and Bittner, 2018]. Apart from the shown methods that work completely on the GPU, there also exist approaches that use this only to generate the auxiliary structure and then optimize the structure on the CPU using a top-down technique [Hendrich et al., 2017].

Besides GPU construction based on Morton codes, parallel clustering algorithms were recently proposed for fast generation on the GPU, such as k-means [Meister and Bittner, 2016] and locally-ordered agglomerative clustering [Meister and Bittner, 2017], which grant comparable and in some cases even higher performance in comparison to algorithms based on Morton codes.

2.3.5 Two-Level Structures

In terms of interactive scenes, oftentimes two-level structures are used. They work by first building an acceleration structure for each individual geometrical object of the scene. Any of the previously presented structures can be used for this. All individual objects are then gathered by an top-level structure. This concept enables rigid object transformations and instancing by only updating the top-level structure without changing the object-structures. This was first introduced with KD-trees [Wald et al., 2003a], but since then also applied to grids [Kalojanov et al., 2011] and BVHs [Parker et al., 2010] [Wald et al., 2014]. One typical disadvantage of BVHs is the overlapping of nodes in the hierarchy, which is especially crucial in two-level approaches with overlapping and colliding objects. A solution for this was proposed, where object-structures were dynamically opened and merged during construction of the top-level structure, which is known as re-braiding [Benthin et al., 2017]. While all previous approaches use the same structure for the top-level as well as the object-level, also a hybrid attempt was proposed, which combines different structures for those levels [Wang et al., 2016].

2.4 Directional/Visibility Structures

Spatial structures are good in subdividing the scene and hence delimiting the number of needed intersection tests by a big amount. However, the subdivision scheme does not include any information of a possible ray's direction. There exist some acceleration structures that additionally incorporate this information and thereby try to grant better scene coverage and higher performance. This is especially used in radiosity calculations, where the mutual coverage of all scene patches is computed [Cohen and Wallace, 1993].

A simple method works by extending the rays starting from a specific point to cones with a given angle, which is also known as cone tracing [Amanatides, 1984]. Thereby, it is possible to include simple image effects in rendering, such as anti-aliasing, soft shadows and glossy reflections. A similar approach gathers multiple rays with the same direction to form beams [Heckbert and Hanrahan, 1984]. Thus, multiple rays are traced at once, which increases coherency and reduces the number of intersection tests, leading to much higher tracing performance. This can be also seen as a predecessor to packet tracing [Wald et al., 2006]. The beams itself can be subdivided in a hierarchical structure and a combination with KD-trees was proposed [Reshetov et al., 2005]. There, the rays inside a beam are able to either skip KD-tree traversal or start from some node inside the tree. The beams themselves can be gathered in a visibility field and the contained objects can be sorted along the beam's direction, which was used to further accelerate the traversal in ray tracing [Mortensen et al., 2007]. However, this results in high memory consumption and initialization time. All these considerations are also useful in terms of acoustic simulation [Laine et al., 2009].

This definition of beams can be used for ray classification in higher-dimensional space [Arvo and Kirk, 1987]. Each ray can be created by a 3-dimensional starting point along with a 2-dimensional parametrization of a unit sphere used as ray direction, therefore resulting in a 5-dimensional space. This space can be subdivided in disjoint volumes, which can be represented as beams gathering all contained rays. For each of those beams the set of scene candidates that are at least partially intersected by the beam is computed and stored in a preprocessing step. This information is then used for ray classification during runtime, where only the candidate set of the ray's corresponding beam is used for intersection calculation. By using a prepocessing step, the number of visible scene candidates within the beams and therefore the total memory consumption can be reduced [Sharpe et al., 2003]. The ray classification scheme was extended to use a

more regular structure based on a equally-sized subdivision of a bounding volume and thereby reduced the 5-dimensional space into a 4-dimensional one based on the starting and ending patch [Kwon et al., 1998]. This was later used in a precomputed sphere projection for ambient occlusion calculation on CPU [Gaitatzes et al., 2008] and GPU [Gaitatzes et al., 2010]. More recently, a similar approach was computed within an octree, where the surface of each node is subdivided in equally-sized regular patches along the center-points of the sides, with connections between each of those patches [Billen and Dutré, 2016].

In terms of radiosity calculation, the simplification of objects to shafts were proposed, which grants more efficient culling strategies through ray-casting [Haines and Wallace, 1994]. In general, a shaft can be constructed by connecting two different planar elements and thereby gathering all rays that go from one element to the other and vice versa. This is used as a starting point for line space calculations. A hierarchical approach was proposed where the two elements can be subdivided and thereby form sub-shafts, which was used in radiosity systems with interactive updates [Drettakis and Sillion, 1997]. There, the term "Line Space" was used in a similar sense as in this work. A somewhat different meaning for this term was used with visibility precomputation in urban scenes. There, a set of visible elements is computed based on a subdivision scheme of horizontal lines [Bittner et al., 2001], which was later enhanced by a vertical parametrization [Leyvand et al., 2003]. Further advancement lead to intersection fields, which can be combined with indirect illumination through photon tracing [Ren et al., 2005].

CHAPTER 3

OVERVIEW - THE LINE SPACE

As explained in chapter 2, suitable data structures are needed for the acceleration of ray tracing. They are mainly divided in two categories: the more commonly used spatial structures, which are used in order to accelerate ray tracing by recursive subdivision of the scene space and thereby reduce the amount of ray-triangle intersection tests, and the less commonly used and more complex directional structures, which oftentimes rely on some sort of ray classification scheme based on beams. This work presents a method to combine the different strengths of these categories by using a simple abstraction of directions and visibilities within bounding boxes, which are the basic building block of most used spatial data structures. There, the visibility is precomputed through ray classification with shaft abstraction within the line space in order to reduce the computation overhead during ray traversal. The final goal is to find the needed intersection information without further traversal calculations but based on extensive precomputation and abstraction to the shafts.

This chapter presents the groundwork on visibility precomputation through shaft abstraction, the line space and its integration within typically used spatial data structures. For this purpose, it is first shortly presented, how it is related to previous work and in which way the previous work is extended by this work. Afterwards, shaft abstraction is explored and beneficial properties of shafts are identified. Based on this, the line space and its main parameters are described.

3.1 Relation and Extension to Previous Work

The main ideas of line space are similar to [Reshetov et al., 2005], where beams of rays are used within a KD-tree in order to reduce the number of cells traversed, but the underlying approach is completely different. In the work at hand, the combination not only works with KD-trees, but builds

on top of bounding boxes and voxels and thereby enables support of all typically used spatial structures, such as BVHs, octrees and uniform grids. Furthermore, instead of using beams as proxy objects for groups of rays, shafts are used, which are formed by combining two separated faces or patches to its convex hull, thus including all rays that start and end on those two faces. Note that shafts were also used in [Haines and Wallace, 1994], however with a more complex meaning. While shafts in their understanding serve as proxy objects to combine two individual geometrical objects each with its own bounding box, a simplified version that just combines two faces is used in the work at hand. These shafts grant special attributes as presented in section 3.2.

Based on shaft abstraction, this work uses precomputed visibility information for the contained rays. This is partially similar to ray classification as proposed by [Arvo and Kirk, 1987]. There, 5-dimensional hypercubes are used as proxy objects for rays, which are created based on a 3-dimensional rectangular starting volume of a ray and a 2-dimensional angle for the ray direction. Visibility information is there precomputed based on these hypercubes. In this sense, the shaft abstraction represents a simplification to this. Instead of collecting and grouping rays by the starting volume and direction, they are grouped according to the two faces of an intersected bounding box, thus presenting a 4-dimensional ray classification similar to [Kwon et al., 1998]. This grouping into shafts was already shown to result in much higher memory efficiency and in this work is additionally shown to be easier usable.

The information that is precomputed in previous attempts is mostly correlating to the information necessary for correct ray tracing results, which is usually the complete list of intersection candidates within the proxy object. During runtime, it is necessary to compute the actual intersecting object via intersection testing with the according ray. Thereby, it is possible to enhance runtime performance at the cost of higher construction time and much bigger memory consumption. In turn, there have been attempts which do not store the complete candidate list for a proxy object, but rather an estimate to the actually contained geometry. This was used for shadow computation [Billen and Dutré, 2016], however an additional traditional data structure was needed for cases where the correct shadow situation could not be concluded through the directional approach.

This work extends the previous work in different ways. A general approach in combining ray classification through shaft abstraction is proposed and used in conjunction with most of the typically used spatial data structures, which is further described with bounding boxes (section 3.2), recursive grids (subsection 4.2.1), BVHs (section 6.3) and two-level struc-

tures (section 7.2). Different types of precomputed directional information are discussed and used within the data structure, which include the pure binary emptiness of the proxy object (subsection 4.2.2) and a representative candidate instead of the complete candidate list (subsection 6.2.1). These were used to enhance performance in multiple parts of rendering via ray tracing through incorporation of approximations, such as empty space skipping for all rays (subsection 4.2.4), acceleration of soft shadow computation (subsection 5.4.2 and with usage heuristic subsection 8.3.2), ambient occlusion calculation (subsection 8.3.3) and indirect illumination (section 6.2), as well as path tracing acceleration (section 7.3).

3.2 Basics of Shaft Abstraction

The basic concept for visibility precomputation is the discrete abstraction and grouping of rays to shafts within a bounding box. In that the precomputed visibility information is generalized for all possible rays within the shaft and later used during runtime. For this, it is now explained how the shafts are generated and which special properties are needed.

The visibility precomputation is done based on voxels or bounding boxes when objects are involved. As voxels can be seen as bounding boxes around a given volume, the term bounding box is used in the following. These boxes may be constructed for each node in the scene hierarchy, but other procedures in constructing bounding boxes can be used as well. Starting from a single bounding box, it is first subdivided in equally-sized subnodes by splitting every dimension in N equal parts. By this, each side of the bounding box surface is subdivided in $N \times N$ subsides (in later chapters also called faces or patches), resulting in $N \times N \times N$ subnodes. Shafts are then constructed by connecting two separate subsides of the surface to result in the convex hull between those. These shafts are used as proxy volumes which contain all possible rays that intersect the bounding box in the two connected subsides. Therefore these subsides can be declared as start and end points of rays in this bounding box. Moreover, the bounding box subsides are countable and can be identified by a unique id. Consequently, the shafts can be uniquely identified by the combination of two subside ids. A LS contains the visibility information for all shafts within the corresponding node.

For easier demonstration, this is illustrated in 2D. Each of the 4 surface sides is subdivided into N subsides, giving a total subside count of $4N$ and a total subnode count of $N \times N$. By connecting every subside with every other subside, a total number of $4N \times 4N = 16N^2$ 2D-shafts are constructed.

	2D			3D		
	# sub sides	**# sub nodes**	**# shafts**	**# sub sides**	**# sub nodes**	**# shafts**
N	$4N$	N^2	$16N^2$	$6N^2$	N^3	$36N^4$
1	4	1	16	6	1	36
2	8	4	64	24	8	576
3	12	9	144	54	27	2916
4	16	16	256	96	64	9216
5	20	25	400	150	125	22500
6	24	36	576	216	216	46656
7	28	49	784	294	343	86436
8	32	64	1024	384	512	147456
9	36	81	1296	486	729	236196
10	40	100	1600	600	1000	360000
11	44	121	1936	726	1331	527076
12	48	144	2304	864	1728	746496
13	52	169	2704	1014	2197	1028196
14	56	196	3136	1176	2744	1382976
15	60	225	3600	1350	3375	1822500
16	64	256	4096	1536	4096	2359296

Table 3.1: Number of shafts for different values of the subdivision parameter N. It is visible that the total number of shafts in 3D ($36N^4$) gets high for big values of N.

This is illustrated in Figure 3.1, where also the shaft connecting the two subsides with ids 6 and 14 is presented. In 3D each of the 6 surface sides of a bounding box is subdivided into $N \times N$ subsides, giving a total number of $6N^2$ subsides, $N \times N \times N$ subnodes and $6N^2 \times 6N^2 = 36N^4$ 3D-shafts. Obviously, the number of shafts gets high with bigger numbers of N, as illustrated in Table 3.1.

The high number of shafts that are constructed in 3D is one of the main downsides in LS application. Therefore, some trivial properties help in reducing the total shaft count in a single LS, as illustrated in Figure 3.2:

Reflexive Shafts formed by similar subsides are degenerated with zero volume and are therefore always empty and omitted, leading to empty values on the diagonal of the LS: $LS(s;s) = \emptyset$.

Collinear Shafts between collinear edges are also degenerated, leading to empty blocks around the diagonal of the LS: $LS(s;e) = \emptyset; \forall s, e$ constructed from the same surface side

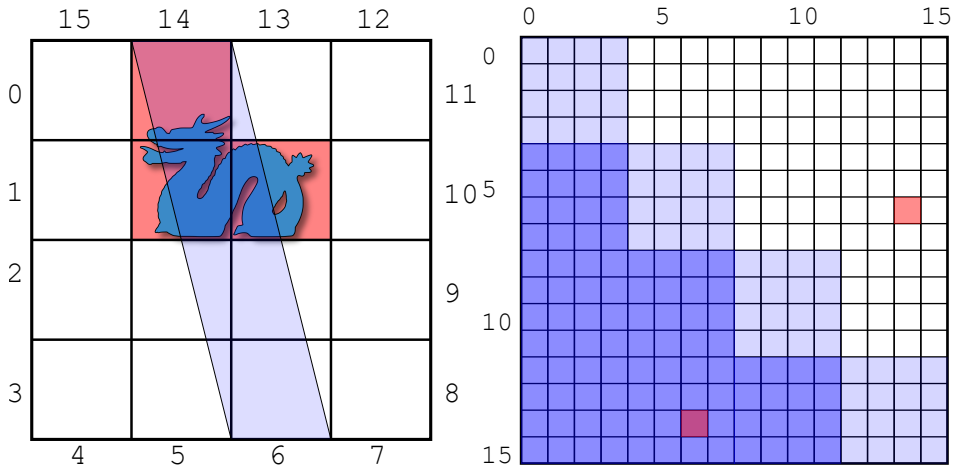


Figure 3.1: Illustration of a single shaft and part of the according LS in 2D. The subsides are identified with numbers from 0 to 15. Each shaft is identified by the tuple of subside ids. One non-empty shaft starting from 6 and ending in 14 with the associated entry in the LS is presented. Light blue entries in the LS represent shafts that start and end on the same sides of the bounding box (collinearity), while the dark blue entries contain duplicate information in the symmetric case.

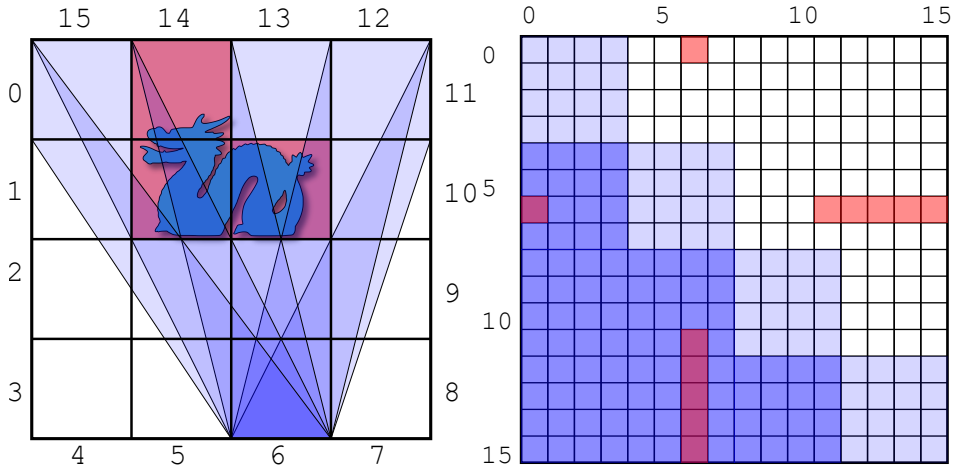
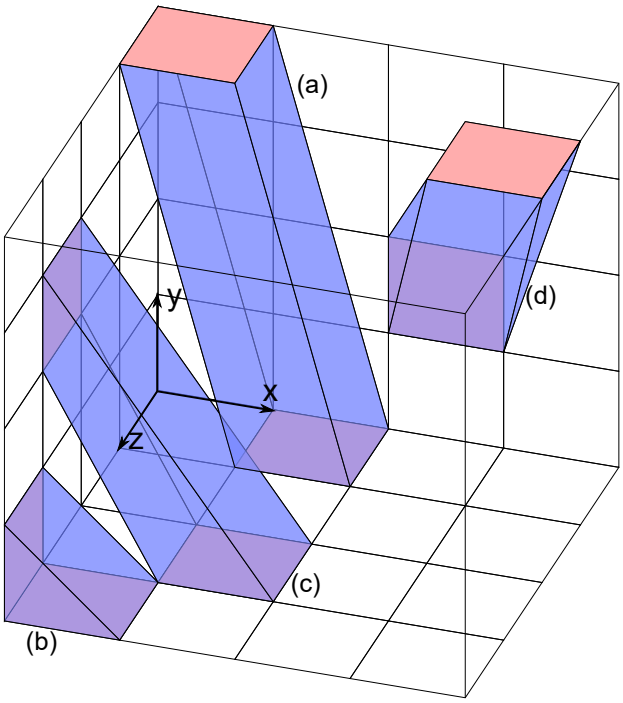


Figure 3.2: Illustration of all non-empty shafts starting from subside 6 in 2D. When the stored visibility information is symmetric, the occupied fields in the LS (shown in red) are mirrored at the main diagonal. As before unnecessary information due to collinearity and symmetry is marked in blue within the LS.



(a): parallel subsides form parallelepipeds

(b): adjacent subsides with an edge connection form right triangular prisms

(c): subsides with center points in parallel alignment with an axis form right quadrilateral prisms

(d): subsides with no special conjunction represent the general case.

Figure 3.3: Different cases of shaft geometry can be distinguished.

Symmetric Shafts may be symmetric depending on the visibility information that is stored within the shaft, then leading to duplicate information in the LS: $LS(s;e) = LS(e;s)$

Using those properties, the total number of shafts can be reduced to less than half. In 2D the number of shafts is reduced by $4N$ due to collinearity and afterwards divided in half to $6N^2$ depending on whether the stored visibility information is symmetric. In 3D the total number is reduced by $6N^4$ due to collinearity and to a total of $15N^4$ with symmetrical data.

Apart from the bounding box subdivision parameter of the LS node and hence the number of shafts for a single node, additional details are needed to compute the total space necessary to store the data structure. That is the amount of memory needed for a single shaft and the total number of LS nodes generated over the complete scene. Both is specific to the application and the way it is integrated within the spatial base data structure and is therefore explored in later chapters. However, in general all applications need to compute some sort of intersection between the scene and the shaft geometry and therefore further information on the actual geometry of a shaft is needed.

A few trivial cases can be distinguished, as illustrated in Figure 3.3. If the subsides that form the shaft are parallel, i.e. they were constructed from opposing surface sides of the bounding box, the resulting geometrical form of the shaft is a parallelepiped. If the subsides have a similar edge and were constructed from adjacent surface sides, the shaft forms a right triangular prism. If the center points of the subsides can be positioned in a plane parallel to an axis plane and are not connected via a similar edge, they form a right quadrilateral prism. In the general case, the subsides are neither parallel nor connected, were constructed from different surface sides and all planes connecting their center points are not aligned in parallel with one of the axis planes. As all trivial cases can be derived from the general case, this one is inspected in more detail.

To achieve simple intersection tests between the scene and the general shaft volume, the shaft geometry needs to be known. It is dependent on the two constructing subsides and all necessary connecting surfaces that form the mantle between those subsides. For simplification, it is assumed that the subsides are quadratic with edge length of 1, therefore a cubic bounding box with width of N is assumed. If the bounding box size differs from this value, a simple scaling has to be executed on the coordinate axis and the intersection points between the ray and the bounding box prior to the following equations. Each subside S can be described by either its four corner vertices $S_{0..3}$ or its bounding box surface side f along with the corner vertex with the smallest axis extents s_0.

$$\begin{aligned} S_{sub} = \text{Quad}(s_0, s_1, s_2, s_3) = \text{Subside}(f, s_0) \\ f \in (+x, -x, +y, -y, +z, -z) \end{aligned} \tag{3.1}$$

The eight corner vertices of both subsides S_A and S_B also represent the vertices of the shaft $S_{A,B}$.

$$\begin{aligned} S_A &= \text{Quad}(a_0, a_1, a_2, a_3) \\ S_B &= \text{Quad}(b_0, b_1, b_2, b_3) \\ S_{A,B} &= \text{Shaft}(a_0, a_1, a_2, a_3, b_0, b_1, b_2, b_3) \end{aligned} \tag{3.2}$$

The subside vertices $S_{0..3}$ are easily computable if the bounding box surface f and the smallest extent vertex s_0 are given. Starting with this corner vertex, the other three vertices can be derived by adding the two spanning axis vectors $\vec{u_s}$ and $\vec{v_s}$ of the bounding box surface f.

$$(\vec{u_s}, \vec{v_s}) = \begin{cases} (\vec{y}, \vec{z}), & \text{if } f \in (+x, -x) \\ (\vec{x}, \vec{z}), & \text{if } f \in (+y, -y) \\ (\vec{x}, \vec{y}), & \text{if } f \in (+z, -z) \end{cases} \tag{3.3}$$

$$\begin{aligned} s_1 &= s_0 + \vec{u_s} \\ s_2 &= s_0 + \vec{v_s} \\ s_3 &= s_0 + \vec{u_s} + \vec{v_s} \end{aligned} \tag{3.4}$$

To generalize the shaft geometry creation, a specific ordering of the shaft vertices can be used. The actual geometrical primitives for the shaft mantle are then created by a combination of the vertices with a fixed arrangement. The vertex ordering is based on the position of the subsides to each other. First, a common spanning axis vector $\vec{d}$ between both subsides needs to be found. With this, the differing spanning vectors $\vec{s}$ and $\vec{t}$ are concluded in such a way, that each of those spanning vectors point away from the other subside. If one of the spanning axis vectors instead points towards the other subside, then the corner vertex with the smallest axis extents s_0 is incremented by the spanning vector and the spanning vector is inverted. Thus, it is guaranteed that without considering the extent in terms of $\vec{d}$ the distance between a_0 and b_0 is the smallest. Following the same premise, i.e. not considering the extent in terms of $\vec{d}$, the connection between $a_0 + \vec{s}$ (i.e. a_2) and $b_0 + \vec{t}$ (i.e. b_2) is the biggest. Additionally, the ordering needs S_A to be lower than S_B in terms of $\vec{d}$. To guarantee this, S_A and S_B along with all vertices are switched if $a_0 \cdot \vec{d} < b_0 \cdot \vec{d}$. Consequently, the distance between $a_0 + \vec{s}$ (i.e. a_2) and $b_0 + \vec{t} + \vec{d}$ (i.e. b_3) is the biggest of all subside vertices. The final assembly is illustrated in the upper left image of Figure 3.4. In the special case where both subsides are on parallel faces of the bounding box, $\vec{d}$ is chosen randomly and $\vec{s}$ and $\vec{t}$ are the same.

$$\begin{aligned} \vec{d} &= (\vec{u_a}, \vec{v_a}) \cap (\vec{u_b}, \vec{v_b}) \\ \vec{s} &= (\vec{u_a}, \vec{v_a}) \setminus \vec{d} \\ \vec{t} &= (\vec{u_b}, \vec{v_b}) \setminus \vec{d} \end{aligned} \tag{3.5}$$

Depending on these values the calculation and ordering of the shaft vertices can be done.

$$\begin{aligned} S_{A,B} &= \text{Shaft}(a_0, a_1, a_2, a_3, b_0, b_1, b_2, b_3) \\ &= \text{Shaft}(a_0, a_0 + \vec{d}, a_0 + \vec{s}, a_0 + \vec{d} + \vec{s}, \\ &\qquad b_0, b_0 + \vec{d}, b_0 + \vec{t}, b_0 + \vec{d} + \vec{t}) \end{aligned} \tag{3.6}$$

The shaft mantle is then produced by two parallelograms $P_{0..1}$ and four triangles $T_{0..3}$. Each of the parallelograms covers one of the side surfaces of the shaft and is connected by two linked points on S_A and S_B along with

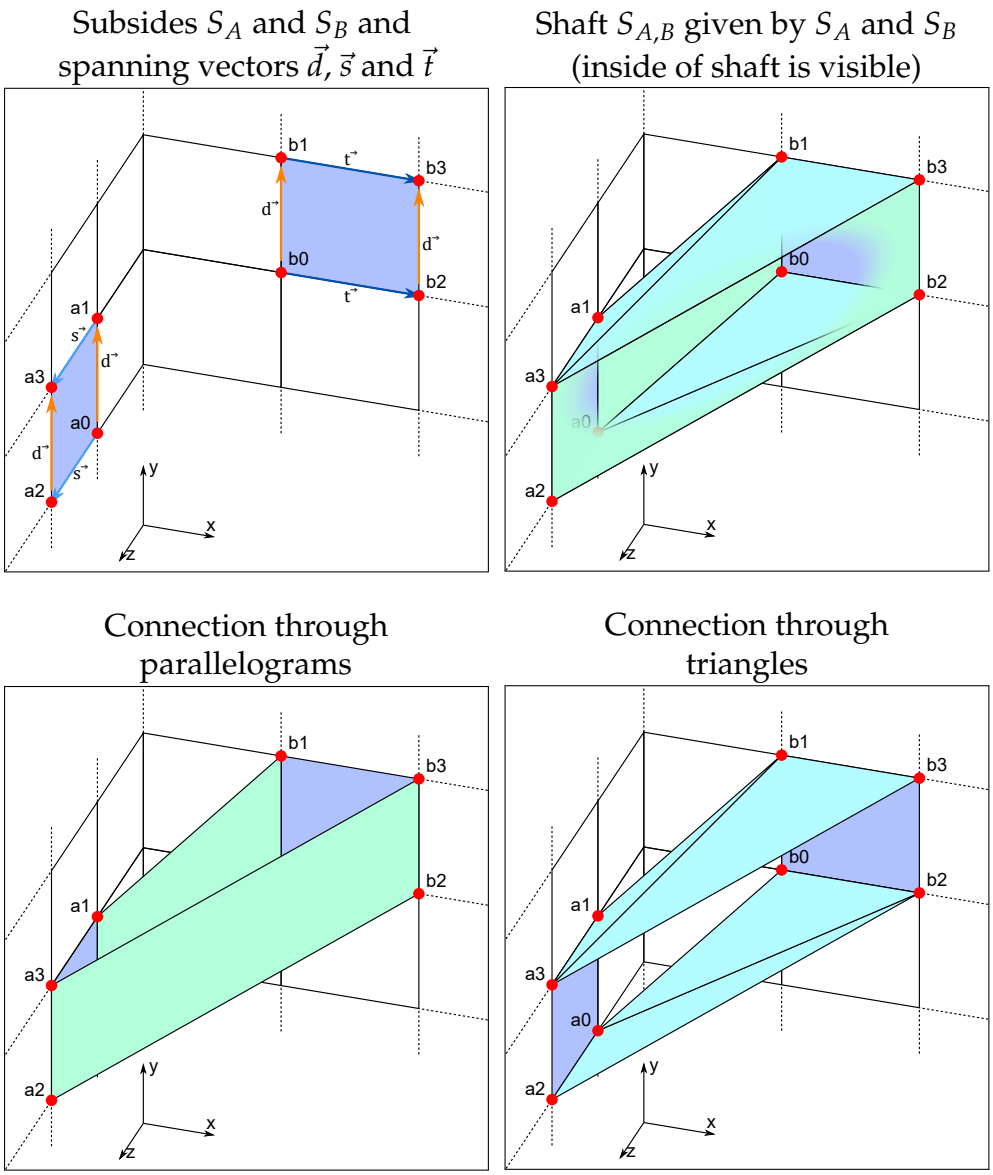


Figure 3.4: Creation of the shaft geometry. Starting with the two subsides S_A and S_B (shown in the top left image), the parallelograms are created by connecting the nearer corner vertices (a_0 and b_0) and the corners shifted by $\vec{d}$ (a_1 and b_1). The same applies to the farther vertices (a_2 and b_2 with a_3 and b_3), as illustrated in the bottom left image. The lower triangles are created by connecting the vertices not shifted by $\vec{d}$ (one triangle with a_0, b_0 and b_2 and second with a_0, a_2 and b_2). The upper triangles connect the shifted vertices (one triangle with a_1, b_1 and a_3 and second with b_1, a_3 and b_3).

the same points shifted by $\vec{d}$. Thereby all four corner vertices are located in one plane and have parallel edges. This is illustrated of the bottom left image of Figure 3.4.

$$\begin{aligned} P_0 &= (a_0, a_0 + \vec{d}, b_0, b_0 + \vec{d}) \\ &= (a_0, a_1, b_0, b_1) \\ P_1 &= (a_0 + \vec{s}, a_0 + \vec{s} + \vec{d}, b_0 + \vec{t}, b_0 + \vec{t} + \vec{d}) \\ &= (a_2, a_3, b_2, b_3) \end{aligned} \tag{3.7}$$

The triangles connect the missing subside edges. As the according vertices are not located in a 3D plane, it is not possible to describe the connecting surface with a planar quadrilateral and instead two triangles are necessary. To conclude which triangles are needed, the property is needed that the participating vertices in the longest connection (i.e. $a_2 = a_0 + \vec{s}$ and $b_3 = b_0 + \vec{t} + \vec{d}$) are already known. Because of this, it can be concluded, that those vertices only participate in a single of the four missing triangles. The result is illustrated in the bottom right image of Figure 3.4.

$$\begin{aligned} T_0 &= (a_0 + \vec{s}, a_0, b_0 + \vec{t}) \\ &= (a_2, a_0, b_2) \\ T_1 &= (a_0, b_0 + \vec{t}, b_0) \\ &= (a_0, b_2, b_0) \\ T_2 &= (a_0 + \vec{d}, b_0 + \vec{d}, a_0 + \vec{d} + \vec{s}) \\ &= (a_1, b_1, a_3) \\ T_3 &= (b_0 + \vec{d}, a_0 + \vec{d} + \vec{s}, b_0 + \vec{d} + \vec{t}) \\ &= (b_1, a_3, b_3) \end{aligned} \tag{3.8}$$

Using the final ordering of the shaft and the definitions for $P_{0..1}$ and $T_{0..3}$, the shaft's geometrical form can be described, as illustrated in the top right image of Figure 3.4 and specifically summarized in Algorithm 2. This convex representation can be used to compute a clipping of scene primitives with the shaft. Thereby intersection tests can be performed efficiently while still producing correct results, which can be used for gathering all candidates intersecting a shaft, similar to [Arvo and Kirk, 1987] and [Kwon et al., 1998]. However, in cases where the stored visibility information is less complex, there may be more efficient techniques in calculating the information. Further development in terms of emptiness precalculation through masking is presented in subsection 4.2.3 and for representative candidate precomputation through ray casting is presented in subsection 6.2.1.

Algorithm 2 The final algorithm to create the convex hull geometry of a shaft given by two subsides S_A and S_B. This covers the general case. Special cases, as illustrated in Figure 3.3, are covered with this algorithm and may lead to parallelograms covering no space or triangles that may be arranged in a single plane and therefore may be combined to planar quadrilaterals.

procedure CREATESHAFTGEOMETRY(*Subside* S_A, *Subside* S_B)
  $(f_A, a_0) \leftarrow$ bounding box surface and smallest vertex of S_A
  $(f_B, b_0) \leftarrow$ bounding box surface and smallest vertex of S_B
  $(\vec{u_a}, \vec{v_a}) \leftarrow$ spanning axis vectors of f_A
  $(\vec{u_b}, \vec{v_b}) \leftarrow$ spanning axis vectors of f_B
    ▷ spanning axis vectors $\in (+\vec{x}, -\vec{x}, +\vec{y}, -\vec{y}, +\vec{z}, -\vec{z})$
  $\vec{d} \leftarrow$ first similar element in $(\vec{u_a}, \vec{v_a})$ and $(\vec{u_b}, \vec{v_b})$
  $\vec{s} \leftarrow \vec{u_a} + \vec{v_a} - \vec{d}$
  $\vec{t} \leftarrow \vec{u_b} + \vec{v_b} - \vec{d}$
    ▷ $\vec{s}$ and $\vec{t}$ similar, if f_A and f_B parallel
  if $(a_0 - b_0) \cdot \vec{s} < 0$ **then** ▷ $\vec{s}$ points towards S_B
    $a_0 \leftarrow a_0 + \vec{s}$ ▷ $\vec{s}$ has to point away from S_B
    $\vec{s} \leftarrow -\vec{s}$
  end if
  if $(b_0 - a_0) \cdot \vec{t} < 0$ **then** ▷ $\vec{t}$ points towards S_A
    $b_0 \leftarrow b_0 + \vec{t}$ ▷ $\vec{t}$ has to point away from S_A
    $\vec{t} \leftarrow -\vec{t}$
  end if
  if $a_0 \cdot \vec{d} > b_0 \cdot \vec{d}$ **then** ▷ S_A "higher" than S_B
    switch a_0 and b_0 ▷ S_A has to be "lower" than S_B
    switch s and t
  end if
  $(a_0, a_1, a_2, a_3) = (a_0, a_0 + \vec{d}, a_0 + \vec{s}, a_0 + \vec{d} + \vec{s})$
  $(b_0, b_1, b_2, b_3) = (b_0, b_0 + \vec{d}, b_0 + \vec{t}, b_0 + \vec{d} + \vec{t})$
  add Quad$(a_0, a_1, a_2, a_3) \rightarrow$ ShaftGeometry
  add Quad$(b_0, b_1, b_2, b_3) \rightarrow$ ShaftGeometry
  add Parallelogram$(a_0, a_1, b_0, b_1) \rightarrow$ ShaftGeometry
  add Parallelogram$(a_2, a_3, b_2, b_3) \rightarrow$ ShaftGeometry
  add Triangle$(a_2, a_0, b_2) \rightarrow$ ShaftGeometry
  add Triangle$(a_0, b_2, b_0) \rightarrow$ ShaftGeometry
  add Triangle$(a_1, b_1, a_3) \rightarrow$ ShaftGeometry
  add Triangle$(b_1, a_3, b_3) \rightarrow$ ShaftGeometry
  return ShaftGeometry
end procedure

3.3 Discussion and Limitations

Obviously, the LS memory consumption as well as its gain in performance significantly depend on the total number of shafts that are created. As demonstrated in Table 3.1, the subdivision parameter N is the main factor in controlling the total number of generated shafts for a single LS node. Additionally, the value of N has significant impact on the size and structure of the generated shafts. A high subdivision parameter leads to long and slim shafts, which are good for precise predictions of the contained scene geometry. However, the number of shafts within the 3D LS is $36 \times N^4$ (or $15 \times N^4$ if symmetrical visibility information is stored per shaft), so a high value of the subdivision parameter results in a huge memory consumption.

Moreover, the number of generated LS nodes is also relevant in terms of memory consumption. This number mostly depends on the used base data structure. The N-tree, as described in subsection 4.2.1, typically has a LS node for each subdivided node in the hierarchy. With the same subdivision parameter N as used for the LS, the number of LS nodes increases with N^3 recursive subdivision. This recursion is typically bound to a maximum depth d of the hierarchy, which hence is the second important parameter for calculation of the total memory consumption. This value is especially important with deep hierarchies such as BVHs, where it is used as maximum depth until which LS nodes are used within the hierarchy. This depth is then also called the maximum integration depth for the LS, as further explored in section 6.3. With a fixed amount of subnodes per subdivided node, it is then possible to compute an upper bound for the maximum total number of generated shafts.

Lastly, the memory size of a single shaft is needed when computing the total memory consumption. In cases, where the total list of intersecting scene primitives is stored as candidates per shaft, as done in [Arvo and Kirk, 1987] and [Kwon et al., 1998], it is not easily possible to compute a single shaft's memory size. However, when only a limited amount of memory is needed for the stored visibility information, the total maximum memory consumption is computable. This is the case in all further chapters. In subsection 4.2.2 a single bit as binary visibility information per shaft is introduced, in order to indicate, whether it is empty or non-empty. In subsection 6.2.1, this is extended to a single representative candidate per shaft. There however, it is not necessary to store empty shafts. Fortunately, those shafts are easily identifiably by the binary shaft information, which leads to an efficient storage of sparsely filled LS nodes, as will be further explained later on in subsection 6.2.2.

Thus, a fixed memory size per shaft can be assumed, and the total number of generated shafts, the total memory consumption, the performance gain and the approximation quality (as is needed in later chapters) mainly depend on the subdivision parameter N and the integration depth d. Hence, these parameters are used in all following chapters. For good traversal results it is mostly stated that the optimal value for N is between 6 and 10 and for d is either 3 or 4. However, these values significantly depend on the type of visibility information stored per shaft and the actual application, for which the LS acceleration is needed.

CHAPTER 4

EMPTY SPACE SKIPPING THROUGH PRECOMPUTED VISIBILITY

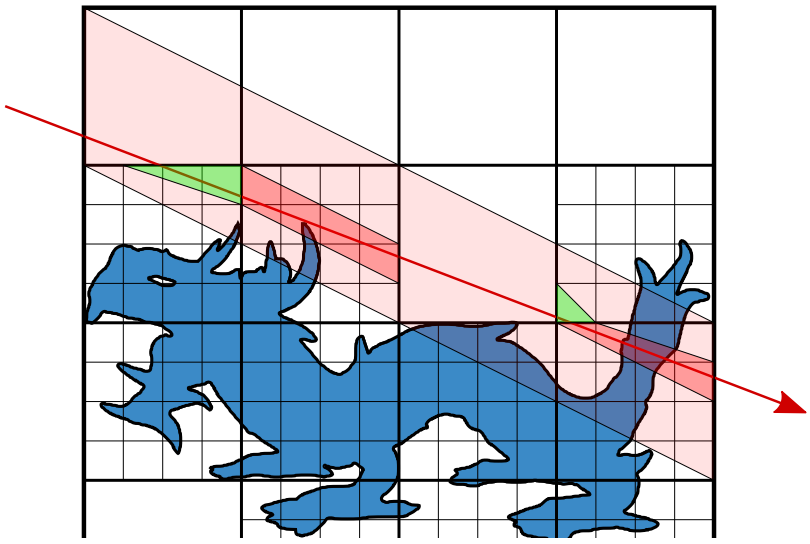


Figure 4.1: Using precomputed shaft visibility information leads to an efficient empty space skipping technique. In this example some of the passed shafts are visualized. Ray segments that pass through empty shafts are skipped immediately (shown in green), while non-empty shafts signal that further investigation is needed (shown in red).

The contents of this chapter are based on the publication:

> Keul, K., Müller, S., and Lemke, P. (2016). Accelerating spatial data structures in ray tracing through precomputed line space visibility. In *Proceedings International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG)*, volume 24, pages 17–25

4.1 Introduction

As explained before, the basic principle of ray tracing is that every visual effect is computed with rays that search for the nearest primitive in a given direction from a known starting point. When this intersection is found, more rays starting from there on are processed. With this it is easy to calculate effects like shadows, reflexions and refractions with only one additional ray for each effect. With additional rays one is able to compute complex visual effects such as ambient occlusion or indirect lighting. However, high image quality results in long rendering times, where even the slightest improvements make significant differences. The main limiting factor is the time it needs to compute the nearest intersection with scene geometry. Therefore, it is important to use an acceleration data structure which supports this task of finding the nearest intersections in an efficient way. Many spatial data structures used today aim to subdivide scene or world space in such a way that the scene is distributed uniformly over every subunit in the data structure.

While this is already a studied field of research, the presented approach goes beyond that. The vision of this work is the precomputation of all necessary visibility information in order to reduce the ray traversal to a single lookup during runtime. As stated before, the needed memory consumption for this is immense and with current hardware not achievable for non-trivial and possibly interactive scenes. However, the presented LS structure is able to use a certain abstraction of precomputation in order to reduce the traversal time significantly. In that context, this chapter presents a first indication of the LS's usefulness with a fast skipping algorithm of empty areas in a spatial base structure.

The goal here is to precompute visibility tests based on directional shafts within the LS and thereby enable the possibility for fast empty space skipping within a traditional spatial base data structure during ray traversal. The precomputations aim to support the base data structure by providing it with an additional condition to decide whether it is possible

to skip the main intersection computations, as illustrated in Figure 4.1. By this, a speed-up in performance of up to 40% compared to the already created base structure is achieved. For the directional precomputation, every possible direction is computed and therefore visibility tests on the whole space with every possible start and end point are enabled. Through this, the speed-up is achieved for all kinds of rays.

Moreover, as the LS is combined with an existing spatial data structure, it is possible to extend the existing traversal algorithms to achieve this speed-up. For optimal usage of the LS, a tree with a higher subdivision parameter compared to the typically used data structures like octrees is generated. It is called the N-tree because of the arbitrary subdivision parameter, which may be dynamically chosen. This is similar to recursive grids, but with the same subdivision for all dimensions. With a higher parameter, it is possible to skip more spatial groups of elements thanks to a single test with this directional data structure.

For a detailed introduction to the necessary related work in ray tracing and existing data structures, it is referred to chapter 2. The main contributions of this chapter are:

- A simple yet effective technique for empty space skipping during traversal using the LS with binary visibility information in its shafts as classification criterion.
- An efficient initialization method based on precomputed subnode masks that rely on the uniform distribution and subdivision of the N-tree.
- An evaluation and parameter discussion in terms of memory consumption and achieved runtime performance of the presented technique.

4.2 The Line Space for Empty Space Skipping

The goal is to extend typical hierarchical acceleration data structures by empty space skipping through precomputated visibility tests based on lines and shafts, which is in some kind similar to ray classification [Arvo and Kirk, 1987]. With this the extended data structure performs just one additional visibility operation per node traversal for a given ray, which is done right before the intersection tests of the ray with the subjects within the current node. If this operation fails, the following intersection tests of the ray and the node subjects are skipped completely. Note that the subjects

of the node are the objects of the scene contained by this node as well as all its own subnodes. Like most acceleration data structures this does not work with dynamic scenes, so the generation of the data structure does not need to be able to compute in real time. For this purpose two-level structures can be used, as discussed in chapter 7. The goal here is to speed up ray tracing of static scenes, and therefore the data structure is only computed once initially.

To clarify the incorporation and benefits of the presented LS technique, the typical traversal of the base data structure is demonstrated first. Following this, it is explained how binary visibility information is used within the LS, how its initialization can be accelerated through a mask-based approach and how this is able to improve the traversal through empty space skipping.

4.2.1 *N*-tree as initial Data Structure

As base data structure the N-tree is used, a variation of recursive grids [Jevans and Wyvill, 1989], which inherently incorporates the LS structure by using the same subdivision scheme. While the octree has 8 subnodes where every edge of the parent node is divided in two equally long parts, every edge of one N-tree node is divided in N equally long parts. It is necessary to have the subnodes equal in size for the visibility test, as this preserves the best shaft accuracy which is explained later on. Therefore, it is not able to use arbitrary splitting planes like in KD-Trees, where different subnodes of one node may differ in size. Although it is possible to store scene objects (the candidates) in every hierarchical N-tree level, performance results suggest that only leaf nodes should contain candidates. Hence, every node of the N-tree is either a leaf node and contains scene objects as candidates or consists of $N \times N \times N$ subnodes.

It is observable that the two main variables, N and the maximum depth of the tree (for further examples d), are chosen arbitrarily and different selections of the values can give similar results. For example, do either $N = 2, d = 6$ (which resembles the typical octree) as well as $N = 8, d = 2$ result in a resolution of 64×64 entries on the deepest hierarchy level. One observable difference lies in memory consumption in sparsely filled trees, where a higher N results in more memory usage due to a higher number of empty subnodes. Moreover, as discussed in section 4.3, these parameters have significant impact on the LS memory size and performance.

The pseudo code of the traversal algorithm for the N-tree is shown in Algorithm 3, which is in principle divided into two parts. First, the

Algorithm 3 The *N*-tree traversal algorithm

```
1:  procedure TRAVERSENODE(Ray r, Node n)
2:      p ← 0
3:      if n has primitives then
4:          p ← nearest primitive intersecting r within n
5:      else if N has subnodes then
6:          while p = 0 and subnodes left do
7:              s ← next subnode in direction of r
8:              if s is non-empty then
9:                  p ← TRAVERSENODE(r, s)
10:             end if
11:         end while
12:     end if
13:     return p
14: end procedure
```

exact start node has to be found. Starting at the root node, the next inner subnode is chosen until the leaf node containing the ray's starting point is reached. After this, the main traversal starts. Every processed node has either candidates, which are tested for intersection with the current ray (lines 3 and 4), or has subnodes, which are recursively processed. All candidates within a leaf node have to be tested for intersection, and if one is found, the traversal stops. Traversing from one node to its subnodes follows a top-down strategy as previously introduced [Revelles et al., 2000] and similar to [Amanatides et al., 1987], the subnodes are traversed in a grid like manner (line 7). If a traversed node neither contains candidates nor own subnodes, it is called "empty". It is not processed at all and therefore skipped (lines 8-10). The algorithm then continues with the next subnode. Traversal stops if a primitive is found within a subnode (line 6).

Subfigure 4.5a shows an exemplary traversal of the *N*-tree. There, the ray starts within the node starting from *S*. At this point, every intersected subnode needs to be checked for intersection, although neither the geometry nor any subnode containing geometry are intersected by the ray. With the proposed LS technique, it is possible to skip those parts immediately, as demonstrated in Subfigure 4.5b.

4.2.2 Visibility Information within Shafts

The LS builds upon the presented N-tree and extends it with an additional visibility test, which decides whether a node is skipped in the traversal. Note that this additional skip condition still works if the node has both, candidates and subnodes. As explained in chapter 3, a node consists of $N \times N \times N$ subnodes. Furthermore, each side of the node's bounding volume is divided in $N \times N$ patches with equal size, which makes a total of $6 \times N \times N$ patches for the subdivided node. These patches are countable and each of them gets its own identifiable index. It is now possible to create shafts connecting every possible index with every other possible index. For each of those shafts it is decidable whether there exists at least one subnode partially or in total within the shaft that contains either candidates or subnodes itself. If a shaft has only empty subnodes, in other words the shaft does not intersect any subnode that is non-empty, the shaft itself is called empty. The LS for a given node contains the information whether a shaft is empty or non-empty for every possible shaft within this node. It is represented as 2D array or texture where the first axis represents the start index of a patch and the second axis represents the patch's end index. Consequently, a pixel with the coordinates x and y denotes the shaft starting at the patch with the index x and ending at the patch with the index y. The value of the pixel represents whether the corresponding shaft is empty or intersects with at least one non-empty subnode.

In the step of deciding whether a shaft is empty, subnodes instead of the actual scene geometry are used due to two reasons. First, the scene geometry is already arranged in the subnodes of the N-tree and possibly quite many of the scene primitives are assigned to just a few subnodes, which significantly increases computation time for those subnodes. Second, the correspondence between non-empty subnodes and encountered shafts can be precomputed, resulting in a mask-based initialization as explained in subsection 4.2.3. Based on this, it is possible to accelerate the LS initialization effectively.

4.2.3 Mask-based Initialization

Using an initialization based on non-empty subnodes instead of actual scene geometry, results in efficient abstraction as the same non-empty subnodes always result in the same LS. This enables the precomputation of masks for every possible subnode within a node, which then can be combined in a mask atlas. Within the initialization step the actual scene geometry is therefore first voxelized to subnodes, while their correspond-

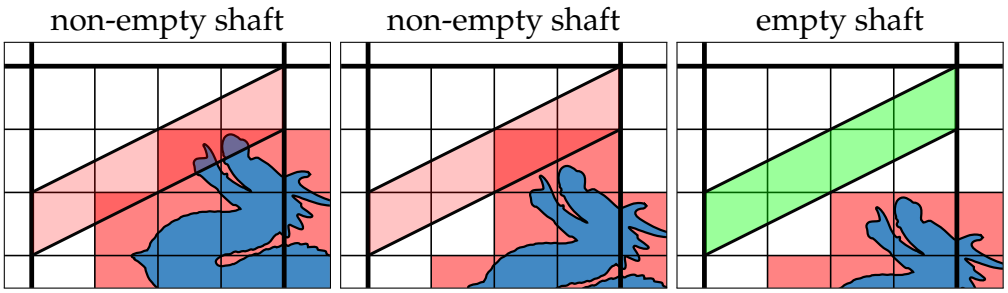


Figure 4.2: LS initialization based on subnodes instead of actual scene geometry. While the left image shaft actually contains geometry and is therefore correctly marked as non-empty, the middle image shaft is marked as non-empty although it does not contain geometry due to the non-empty subnodes it contains. The shaft in the right image is correctly marked as empty, as it does not contain non-empty subnodes and therefore also contains no scene geometry.

ing masks are then used to create the current node's LS. The mask atlas consists of N^3 masks (one for each possible subnode with a subdivision parameter of N per dimension), which themselves contain the correspondence information for $15N^4$ shafts. Hence, the atlas has $N^3 \times 15N^4$ entries in total. This usage enables the initialization without the need of actually testing the geometry for intersection with the shafts and therefore increases generation performance significantly. However, before the mask atlas can be used, it has to be constructed. The initial mask atlas generation can be done by calculating intersections between subnodes and all possible shafts of the node. For this, the convex shaft geometry described in section 3.2 can be used along with a clipping algorithm. This process for constructing the mask atlas can be done once for each necessary value of N and the results can be permanently stored and reused when needed.

One drawback of this subnode-based construction scheme is the incorporation of a degree of inaccuracy, caused by the node abstraction. This might lead to some cases, where a subnode covers a shaft and only contains scene primitives that do not intersect the given shaft. Then the shaft is wrongly marked as non-empty due to the intersection with the subnode, although none of the primitives actually intersect the shaft. Consequently, this reduces the gained effectiveness of the empty space skipping by a small amount but increases initialization tremendously. This is illustrated in Figure 4.2.

Figure 4.3 shows the mask and therefore the relevance of one non-empty subnode (marked in red) to the LS. The left side demonstrates which shafts

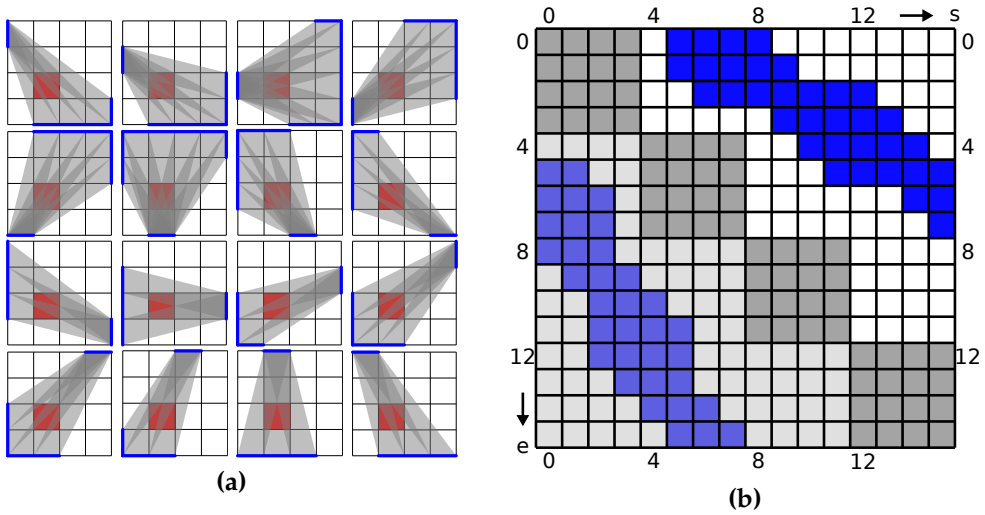


Figure 4.3: (a) All shafts covering one subnode (red) in 2D and (b) the resulting LS (bit mask). Every shaft with start index x and end index y fill the corresponding pixel in the LS. Symmetry and collinearity of the LS are quite obvious.

are marked as non-empty because of the given subnode for each possible start index. The right side shows the corresponding LS where exactly those pixels are marked that belong to non-empty shafts. Note that the outcome of this LS is always the same for the given subnode, independently from the actual size, orientation or count of the scene primitives contained by the subnode. This demonstrates the significance of a precomputed mask, which can be used for every subnode.

The pseudo code for the generation algorithm for all LSs of every node in the N-tree is shown in Algorithm 4. The presented approach works in a top-down way starting with the root node. A LS for a node is only necessary, if the node itself contains subnodes. Every LS is computed with the help of the mask atlas. For every non-empty subnode of a node all entries of the corresponding masks are combined and result in the LS of the current node (lines 4-6). In the binary case, where it only matters whether a shaft is empty or not, this combination is done with a simple ”or” operation for every entry of the mask with the corresponding entry in the LS. The LSs of every subnode consisting of subnodes itself are then computed recursively (lines 7-9).

Figure 4.4 presents an example for a 3D LS. As with the previous examples, N is set to 4. It is obvious that the LS is much more complex compared to a 2D LS. Where in the 2D case every side is subdivided in 4 smaller parts, making it a total of 16 subsides, in the 3D case every of the 6

Algorithm 4 Calculation of LS, starting in the root node. All masks are gathered and combined to the actual LS for this node. Afterwards the procedure is called recursively.

```
1:  procedure CalcLineSpace(Node n)
2:      LS ← create LineSpace for node n
3:      for all subnodes s ∈ n do
4:          if s is non-empty then
5:              mask ← mask denoted by s in maskatlas
6:              AddMaskToLineSpace(LS, mask)
7:              if s has subnodes then
8:                  for all subnodes c ∈ s do
9:                      CalcLineSpace(c)
10:                 end for
11:             end if
12:         end if
13:     end for
14: end procedure
```

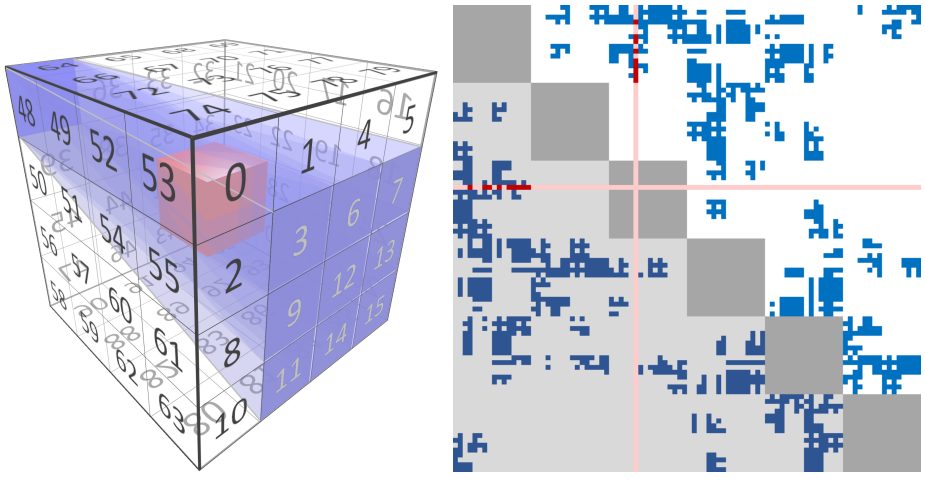


Figure 4.4: (a) All shafts in 3D intersecting the red subnode from the start index with index 37. (b) LS bit mask (4^3 subnodes with 962 LS-entries) for the red subnode. Note that the subnode itself can be subdivided as well and can therefore include its own LS.

bounding sides is subdivided in 16 patches, and therefore, making a total of 96 possible start and end sides. Subfigure 4.4b shows the mask for one subnode (marked in red). For every start index from $s \in 0..95$, a one bit entry provides the information, whether the shaft to end index $e \in 0..95$ intersects this subnode. In the shown example, there are 9 resulting shafts for the starting patch $s = 37$, which can be seen in the red column of the LS.

4.2.4 Traversal and Early Ray Termination

The traversal of the LS is mostly equivalent to the N-tree traversal. In fact, the presented algorithm is just extended by another skip condition, which is added before the subnodes are processed (after line 5 in Algorithm 3). The skip condition checks, whether the LS entry corresponding to the current node is marked and therefore performs a simple kind of ray classification. If this is not the case, it means that all subnodes within the current shaft are empty, the current ray is classified as non-blocking within the shaft and no subnode needs to be processed with the ray. The shaft itself is determined by the precise start and end index within the node, which are intersected by the ray. These have to be computed first in order to identify the shaft the current ray belongs to.

Figure 4.5 presents an exemplary traversal using the LS. For a given ray, the intersection with the root node is computed in order to determine the initial start index S and end index E. The x-, y-, z-coordinates of S and E are mapped to side indices of the root node's surface, yielding the indices for the top level LS. In the example, the big shaft contains non-empty subnodes. Therefore, the subnode covering the ray origin is selected and from there on the traversal of the subnodes starts similar to the N-tree traversal. If one of the inner nodes is not subdivided, the candidate list of this node (if any) is checked for intersection and if no intersection is found the traversal continues with the next inner node. If a node is subdivided, the next level LS is checked first with new start and end indices. If the shaft is not empty, the traversal is proceeded with smaller increments. In the example all green shafts indicate that there are no non-empty inner subnodes, and therefore, these inner subnodes are skipped at all.

4.3 Results and Discussion

The presented method was implemented in C++, exploiting SIMD operations (SSE) and multi-threading on a CPU. The results were evaluated on a PC with AMD Phenom II X6 1090T (6 cores, 3.5GHz) and 16 GB DDR3

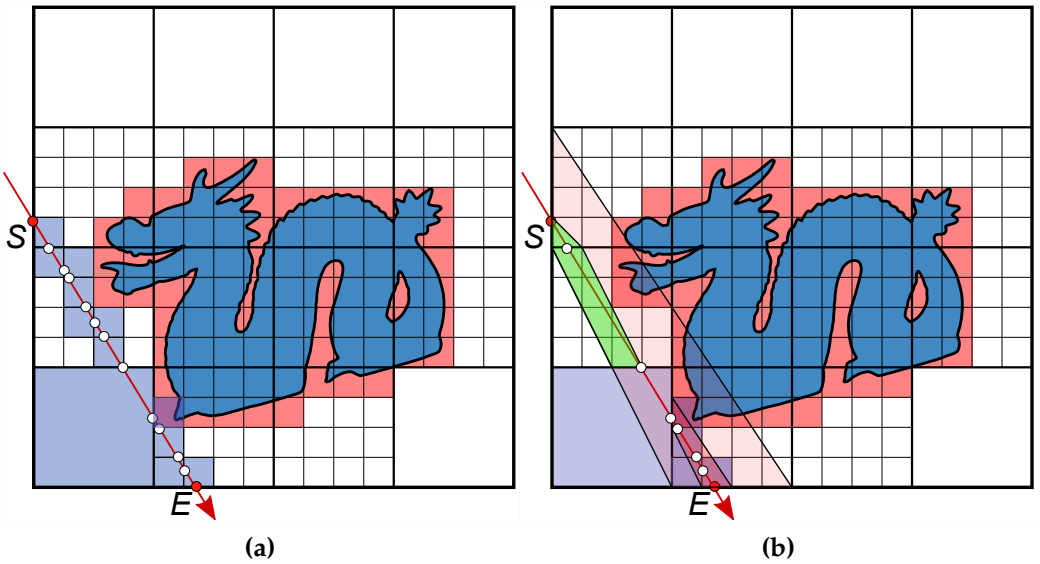
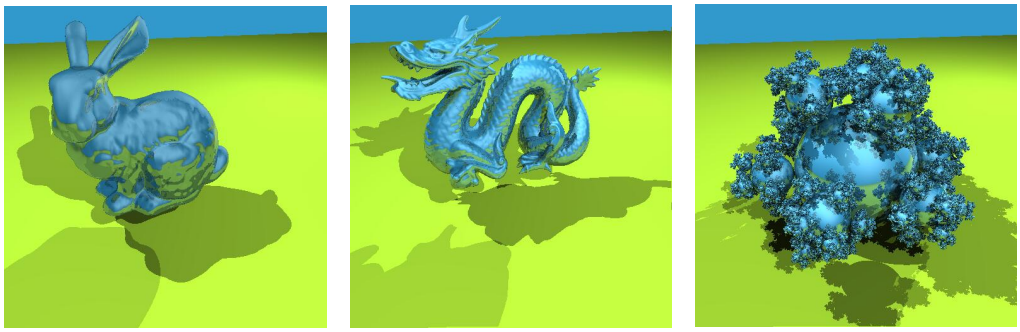


Figure 4.5: Different traversal strategies for *N*-tree without and with LS usage. (a) Typical traversal of the *N*-tree. Although no subnode containing geometry (red) is intersected by the ray, the algorithm traverses every possible subnode (light blue) intersected by the ray. (b) Traversal with the LS. Instead of testing every subnode intersected by the ray, it is first checked if the corresponding shaft intersects any non-empty subnodes. If this is not the case (like shown with the green shafts), all subnodes within the shaft are skipped.

RAM. The used ray tracer computes intersection points for primary rays and up to 10 levels of reflections, where every primitive of the scene geometry is reflecting the ray. For every intersection with scene geometry 3 light sources are used for lighting and for each of those one shadow ray is evaluated. By using reflections and shadow rays, incoherent rays are used, which are traced by the presented method with no difference in comparison to coherent rays. All scenes were rendered with a resolution of 512×512 using different camera angles. The result is the average run time.

Multiple well-known test scenes with different characteristics and of different size of primitives have been used for evaluation (shown in Figure 4.6). Those are divided in scenes showing individual objects only (Bunny and Dragon), architectural scenes (Dubrovnik sponza and Conference room) and a fractal scene (sphere flake using spheres instead of triangles). The individual objects represent the quality of the data structure for a single object only, where many primitives are concentrated in small space. For this purpose the Bunny is a model with a rather low number of primitives, whereas the Dragon consists of a lot of primitives. The architectural scenes



BUNNY
(69k tris)

DRAGON
(871k tris)

SPHEREFLAKE
(597k spheres)

DUBROVNIK SPONZA
(66k tris)

CONFERENCE ROOM
(331k tris)

Figure 4.6: Test scenes used for the performance measurements. Those include individual objects with a varying number of triangles (Bunny and Dragon), a fractal scene using spheres instead of triangles and architectural scenes with different number of triangles (Dubrovnik sponza and Conference room). The images were rendered using 3 light sources and multiple levels of reflection.

represent conventional scenes, which may for example be used in games or films. The sponza is used as a scene with few primitives and the Conference room as scene with quite many primitives. The sphere flake is a fractal scene, which consists of a lot of primitives (spheres in the shown example). Those primitives are not concentrated in the center of the object, but are equally distributed.

For the N-tree and the LS the size of the data structure and the runtime performance within the ray tracer are evaluated. Those are compared with the standard implementations of the uniform grid and the octree to show that the use of visibility information is an improvement of typical well-known spatial data structures. Furthermore, the two main parameters of the N-tree and the LS, the subdivision parameter N and the maximum

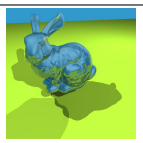
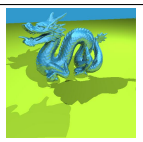
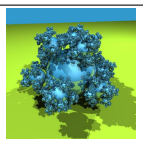
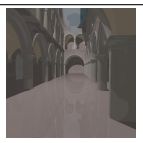
Scene			UG	OT	NT	LS
	BUNNY (69k tris)	**params**	128^3	d $\rightarrow$ 7	(9,3)	(9,3)
		perf (fps)	9.01	7.30	8.13	9.90
		size (MB)	78.4	55.2	82.5	106.7
	DRAGON (871k tris)	**params**	256^3	d $\rightarrow$ 9	(7,4)	(7,4)
		perf (fps)	3.06	3.01	3.37	3.95
		size (MB)	441.0	438.1	823.2	929.6
	SPHEREFLAKE (597k spheres)	**params**	128^3	d $\rightarrow$ 7	(8,3)	(8,3)
		perf (fps)	6.62	4.81	5.59	7.75
		size (MB)	200.8	187.9	511.6	644.0
	SPONZA (66k tris)	**params**	128^3	d $\rightarrow$ 10	(10,3)	(10,3)
		perf (fps)	0.82	0.56	0.71	0.84
		size (MB)	80.4	55.1	220.0	294.7
	CONFERENCE (331k tris)	**params**	128^3	d $\rightarrow$ 10	(10,3)	(10,3)
		perf (fps)	0.72	0.63	0.77	0.92
		size (MB)	213.3	190.8	236.8	249.9

Table 4.1: Empty space skipping evaluation for the test scenes shown in Figure 4.6. All renderings contained 3 light sources and up to 10 levels of reflection. Typical spatial data structures (Uniform Grids (UG) and Octrees (OT)) are compared with the N-tree without and with LS usage (NT and LS, Parameter sets as (N, d)). Only the best parameter sets in terms of traversal time are shown. Performance is measured in frames per second, including all primary rays and reflections, memory size is presented in MB.

tree depth d, are varied and the differences in size and performance are investigated.

The results of the tests are shown in Table 4.1. Several parameter sets for all data structures are evaluated and only the best for each scene were considered. Note that the value of d belongs to the maximum tree depth of the data structure, which is not always needed. In scenes with a small number of primitives it is therefore possible that a big value of d does not provide any benefit.

The uniform grid grants good performance, especially in rather small scenes. The memory size used is in all test cases among the smallest. The optimal resolution for the uniform grid in most test scenes is 128^3 voxels

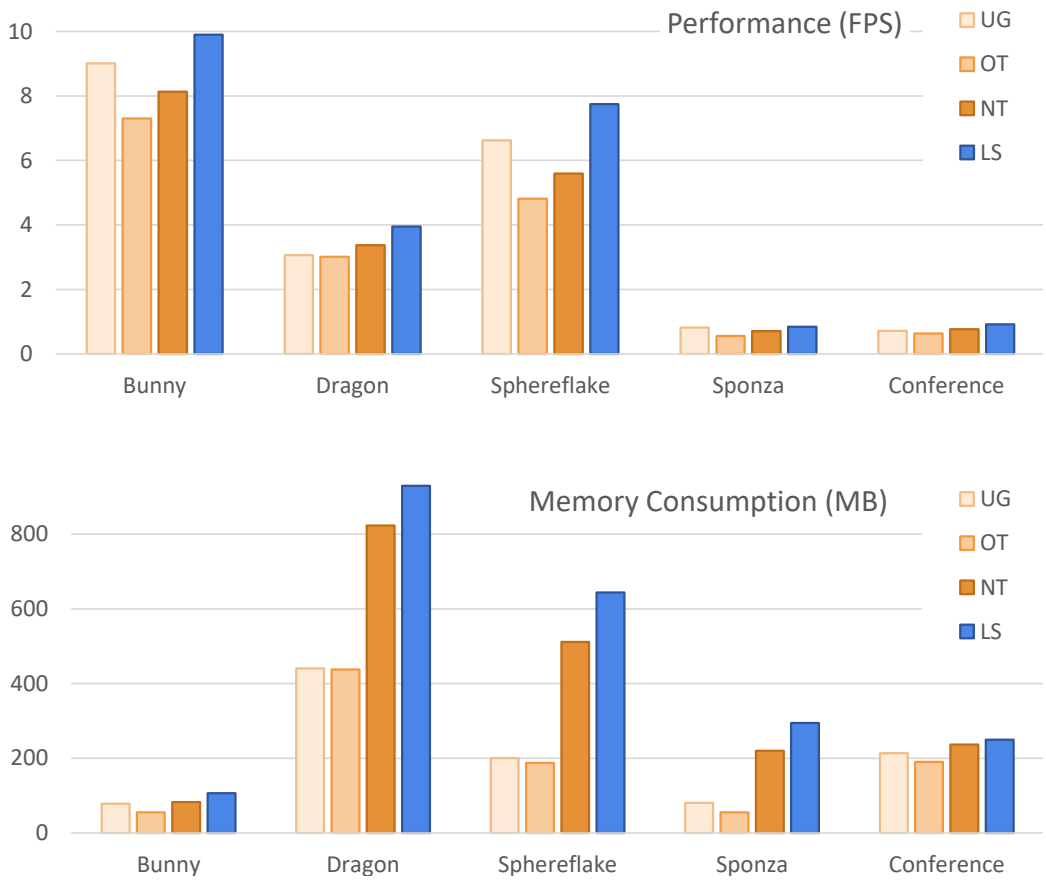


Figure 4.7: Illustration of the performance values (in frames per second) and the memory consumption (in MB, from Table 4.1). The *N*-tree (NT) consumes more memory in comparison to the Uniform Grid (UG) and the Octree (OT). Line Space usage (LS) further increases the needed memory but yields best performance in all cases.

in total. A higher resolution results in a higher traversal cost and a much higher memory consumption of the grid structure and might therefore only be beneficial in big scenes (like the dragon). In comparison to the uniform grid, the octree has a higher performance in those big scenes (dragon and conference room), but worse in small scenes. The memory consumption is dependent on the value of d, where a small value results in a smaller memory consumption. In big scenes a big value of d is beneficial for performance but unfavorable for the required memory size.

The *N*-tree has a higher performance than the octree due to the higher subdivision parameter, where every node is traversed in a grid-like manner. In most cases the *N*-tree performs better than the uniform grid, especially

in the architectural scenes or in scenes with a high number of primitives. This is demonstrated in Figure 4.7. While a high value of N grants better performance, the higher subdivision parameter results also in a bigger memory consumption, especially in sparsely filled N-trees, as also shown. If a node is subdivided, it results in quite a lot of subnodes (N^3), even if only a few of them are actually needed. The optimal parameter sets of the N-tree in respect to the performance have been achieved with a value of N between 6 and 10. The optimal value of d is mostly either 3 or 4.

The LS, as an extension to the N-tree, is always beneficial. In all test scenes the optimal parameter sets were the same as for the conventional N-tree. Obviously, the additional usage of the LS results in a bigger memory consumption. However, due to the fact that only non-leaf nodes need a LS, this increment in memory size is quite acceptable in comparison to the total required memory size. While high values of N and d are a disadvantage in terms of memory consumption, they are beneficial for traversal performance. A big value of N leads to long but slim shafts referring to many but small subnodes. If the shaft is empty, it therefore allows for a quick skip of many subnodes in just one computation. Moreover, long and slim shafts contain small subnodes. Even if these subnodes are intersecting the shaft only for a small part, the amount of subnode space outside of the shaft is just small in comparison to the length of the shaft. In comparison to the uniform grid and the octree, the LS usage results in the highest performance. However, it also has the highest memory consumption, as shown in Figure 4.7.

An important observation is that the traversal performance and the memory consumption significantly depend on the values of the subdivision parameter N and the maximum tree depth d. While the table shows only the best values for every data structure, it is observable that the results are different for the cases where the values for all data structures lead to the same resolution. One example for those values are a resolution of 512^3 for the uniform grid, a maximum tree depth of 9 for the octree and the values $N = 8$ and $d = 3$ for the N-tree and the LS. In those cases the size of the data strucutre and the performance of the traversal are way higher for the N-tree in comparison to the uniform grid and the octree.

The main benefit of the N-tree comes with the usage of the LS. For this, the performance gain of the LS in comparison to the N-tree is evaluated for different values of N and d. The results are shown in Table 4.2, where a performance gain of the LS of up to 40% (1.4×) in comparison to the N-tree is achieved. However, this comes at the cost of memory consumption, where a high value of N is especially bad, because of the high number of possible shafts ($15N^4$) for every subdivided node. The corresponding

Parameters		NT	LS	Δ
$N \rightarrow 5$,	**Perf (FPS)**	2.92	3.00	+2.7%
$d \rightarrow 3$	**Size (MB)**	40.3	40.8	+1.2%
$N \rightarrow 5$,	**Perf (FPS)**	7.30	8.13	+11.4%
$d \rightarrow 4$	**Size (MB)**	57.1	60.3	+5.6%
$N \rightarrow 5$,	**Perf (FPS)**	7.35	7.94	+8.0%
$d \rightarrow 5$	**Size (MB)**	57.6	61.9	+7.5%
$N \rightarrow 6$,	**Perf (FPS)**	5.05	5.56	+10.1%
$d \rightarrow 3$	**Size (MB)**	42.7	44.2	+3.5%
$N \rightarrow 6$,	**Perf (FPS)**	6.94	7.94	+14.4%
$d \rightarrow 4$	**Size (MB)**	96.9	112.2	+15.8%
$N \rightarrow 6$,	**Perf (FPS)**	6.90	7.94	+15.1%
$d \rightarrow 5$	**Size (MB)**	96.8	112.4	+16.1%
$N \rightarrow 7$,	**Perf (FPS)**	6.76	7.63	+12.9%
$d \rightarrow 3$	**Size (MB)**	47.5	52.4	+10.3%
$N \rightarrow 7$,	**Perf (FPS)**	6.94	7.87	+13.4%
$d \rightarrow 4$	**Size (MB)**	109.5	132.8	+21.3%
$N \rightarrow 8$,	**Perf (FPS)**	6.62	8.47	+27.9%
$d \rightarrow 3$	**Size (MB)**	56.8	70.5	+24.1%
$N \rightarrow 9$,	**Perf (FPS)**	8.13	9.90	+21.8%
$d \rightarrow 3$	**Size (MB)**	82.5	106.7	+29.3%
$N \rightarrow 10$,	**Perf (FPS)**	6.02	8.93	+48.3%
$d \rightarrow 3$	**Size (MB)**	123.3	172.8	+40.1%

Table 4.2: Performance comparison between the N-tree without and with the usage of the LS (NT and LS) for different parameter sets of N and d. Higher values result in bigger memory consumption but smaller traversal time with the usage of the LS. The used scene is the Bunny as individual object, other scenes produce similar results.

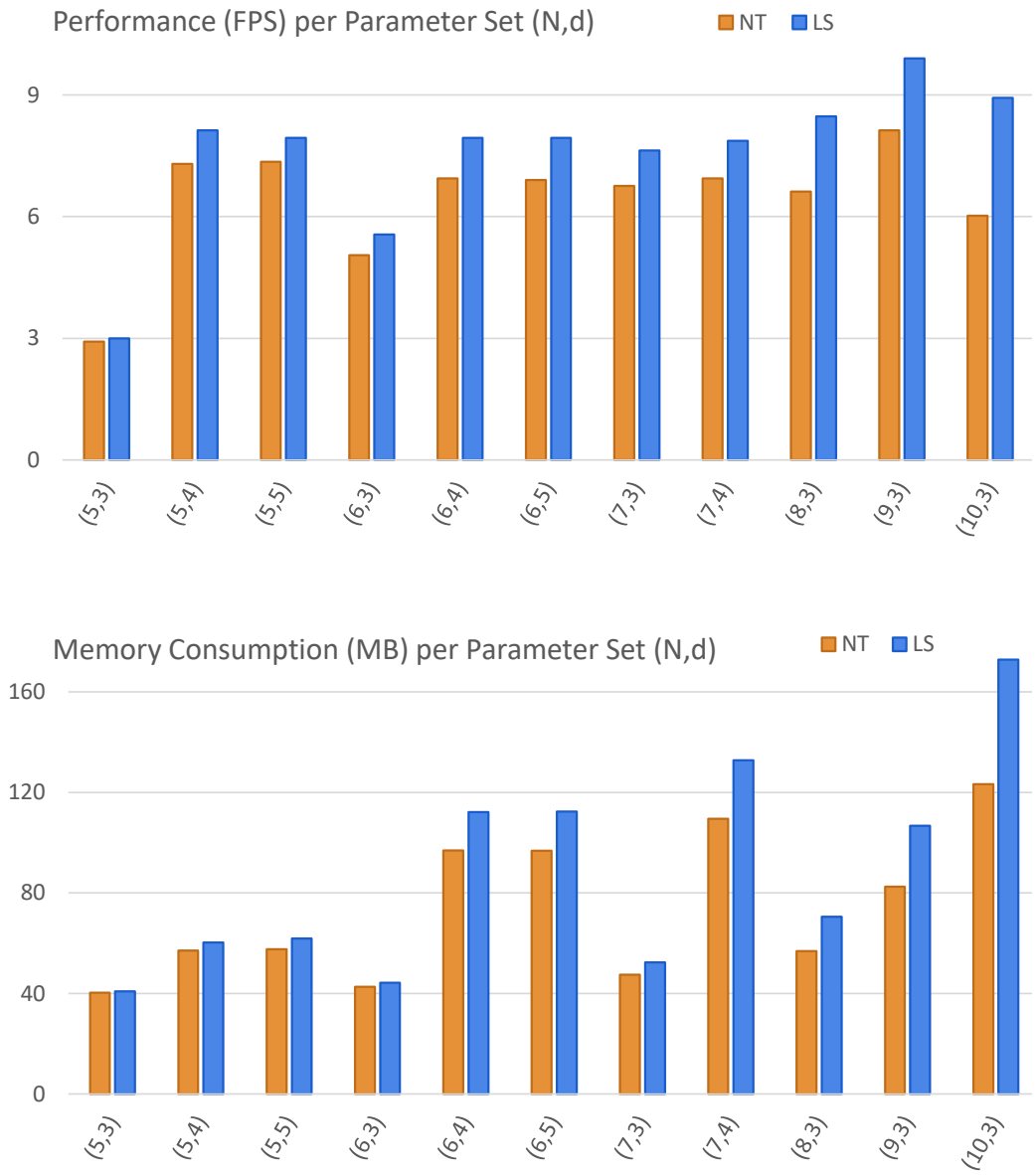


Figure 4.8: Illustration of the performance values (in frames per second) and the memory size(in MB, from Table 4.2) for different parameter sets of the N-tree without and with Line Space usage (NT and LS). As above, N represents the subdivision parameter and d represents the maximum tree depth. The Line Space usage improves performance in all cases, but as it needs to be stored in addition to the base data structure, the needed memory is increased.

memory consumption of the different parameter sets as well as the achieved performance is also illustrated in Figure 4.8. The evaluated test scene is the Bunny, but the results for the other scenes are similar and indicate the same results. In all test cases it is observable that the usage of the LS for a small value of N ($N < 5$) brings little to no benefit. The same applies to big values of N ($N > 10$). As explained above the reason for the former is that the shafts are wider if N is small and the amount of subnode space outside of the shaft is bigger in comparison to its length. While this is unproblematic for long shafts with a big value of N, the problem there is that the shaft loses the potential of prediction because of its length. One non-empty subnode is sufficient to mark the corresponding shaft, so that the traversal needs to check all subnodes. It is observable that big values of d may not make any difference in performance. The reason for this is that the geometry is sufficiently stored in higher nodes and the maximum tree depth of d is not needed. Moreover, if the values of N and d are too big ($N > 10, d > 5$) the data structure is too memory consuming and not usable. The benefit of the LS as well as the optimal choice of the parameters are scene dependant, but in all choices of parameters the usage of the LS results in higher performance than the corresponding N-tree without LS.

4.4 Conclusion and Future work

A novel and effective extension to existing spatial data structures is presented. First, the N-tree, a variation of the Octree, has been discussed. Based on this, the LS is introduced as an advancement for the N-tree by taking directional visibility information into account. Algorithms for the generation and improved traversal were shown. By using binary information for the possible emptiness of all shafts within one node, it is concluded whether it is necessary to test subnodes of the current node or if it is possible to directly skip them due to empty space detection. This additional skip condition results in a notable speed-up of up to 40% in runtime performance for all shown test cases and thereby the general usefulness of the LS for tracing acceleration through empty space skipping was shown. Consequently, there exist multiple directions for further study.

The binary entries in the LS are enough for estimating whether a ray from one point to another might be intersected by scene geometry. With this information it is possible to compute approximated shadows without testing the scene geometry for intersection at all. This might be sufficient for shadow computations of non-primary rays. Even for primary rays the resulting error may become negligible with a high value of d. This

technique might also be used in rasterization where the computation of soft shadows is a rather tough topic. An obvious option for faster generation or better runtime performance is to port the data structure and the traversal to latest generation GPU architectures since many necessary tasks could benefit from parallel computation. Further investigation in LS-produced shadows and appropriate GPU structures is presented in chapter 5 and chapter 8.

Another attempt is to not only store binary visibility information in the LS, but more complex data. By using an incremental counter instead of the binary entries within the shafts, the data structure may be updated during runtime and therefore may possibly be fast enough to handle dynamic scenes in realtime. The counter is changed for each object intersecting a shaft. Thus, the LS can be rebuilt efficiently by decrementing the counter if geometry is removed and by incrementing it if geometry is added. A different approach to achieve the support of dynamic scenes is to use two-level structures, where the LS is generated on object-level and an upper-level structure gathers and combines these object structures. This would enable a simple yet efficient method for rigid object transformations and geometry instancing. Further development to this is presented in chapter 7.

Another attempt concerning more complex shaft information is to store a list of candidates directly within shafts instead of storing them in the nodes of the N-tree. This would result in several advantages. First, the candidates within the shaft may be sorted beforehand and thereby improve performance during runtime. Moreover, the traversal itself may work without the typical node structure based on voxels but rather based on shafts, which is more accurate and efficient. However, it is expectable that the storage of the complete candidate list within shafts leads to a tremendous increase in memory consumption. A promising trade-off between usability and memory usage is presented in chapter 6 where a single scene primitive is stored as representative candidate per shaft, which is then used in terms of fast indirect illumination approximation.

Right now, the LS as directional visibility data structure is able to improve typical recursive grid structures. An important step is to adapt the LS structure to a wider field of spatial acceleration structures. By abstraction of the directional information to bounding boxes, many of the currently used structures may benefit from the precomputed visibility information. Further research in this topic is presented in chapter 6, where the LS adaption to bounding boxes and therefore the usage with state-of-the-art data structures like bounding volume hierarchies is shown. There, it is demonstrated, that the gain in performance through empty space skipping can also be applied to other structures.

CHAPTER 5

APPROXIMATION OF SOFT SHADOWS THROUGH PRECOMPUTED VISIBILITY

Figure 5.1: Soft Shadow computation by using 49 Shadow rays for a static scene. The scene is rendered with typical forward rendering while the shadows are computed with a ray tracer using early LS termination. The usage of the LS grants better performance compared to an equivalent BVH-based ray tracer with similar quality.

The contents of this chapter are based on the publication:

> Keul, K., Klee, N., and Müller, S. (2017). Soft shadow computation using precomputed line space visibility information. *Journal of International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG)*, 25:97–106

5.1 Introduction

Up to this point, the LS was introduced as effective method for empty space skipping in chapter 4. This was achieved with binary visibility information stored within the shafts, indicating whether a shaft is empty or non-empty. To obtain this information during runtime, only a single request of the LS was needed per shaft. With this, the previous chapter demonstrated the usefulness of visibility precomputation based on the LS. Nevertheless, until now only the empty parts during traversal are accelerated, non-empty parts are traversed in a traditional way with possibly lots of intersection tests. Based on the LS with binary visibility precomputation, this chapter presents a simple technique for fast yet approximated soft shadow computation that is done completely without intersection tests during runtime.

In terms of rendering, computing shadows is one of the most important ways to enhance realism in a scene. Shadows increase the spatial perception and with that the overall appearance of realistic rendering results. A simple way to compute shadows is to compute the distance of the foremost objects to a point light source first and store those in a shadow map. This map is then used to differentiate occluded from lighted objects. This approach may be applied in combination with typical rendering and is therefore useful for exploitation of the parallel nature of the graphics processing unit (GPU). With that it is fast and gives realistic results for point light sources. But in most cases area lights are favored because of better quality in realistic scenes and shadow mapping techniques fail to produce fast shadows of those with good quality. Therefore, in most approaches some sort of ray tracing method is used to approximate the surface of the area light source with multiple samples. While the results of these techniques have a good and physically plausible quality, the computation needs quite a lot of time even with good traditional acceleration data structures.

A novel approach, presented in this chapter, is to compute approximate shadows using the LS as acceleration data structure. By using the LS as presented in chapter 4, it is possible to precompute visibilities, which are used in this method for blocker calculation. With this, it is not needed

to test the actual scene geometry for intersection, but use approximate occlusions based on the shaft information of the LS. This increases the rendering performance and allows to only store the LS data structure in GPU memory with no need of storing any geometry information at all. One downside of the presented method is that the produced shadows are not precise because of the approximations with shafts. By using the LS for area lights, it is possible to show that these inaccuracies are negligible. The results demonstrate that the LS usage leads to a faster method compared to previous ray tracing data structures with a performance improvement of up to $3\times$ and better quality compared to typical image-based techniques.

The main contributions are:

- A fast but approximate technique for soft shadow computation of static scenes in real-time using the precomputed visibility within shafts of the LS as termination criterion.
- An efficient structure based on data pools for storing and using the LS with binary visibility information on modern GPU architectures and thereby benefit from significant acceleration due to parallelization.
- An evaluation in terms of achieved runtime performance and image errors of this technique.

5.2 Related Work in Shadow Computation

The main topic of this chapter is the tracing of precomputed directional shadow information in a spatial data structure. Therefore, the most relevant work in real-time shadow generation is presented here. For an introduction in typical spatial and directional ray tracing acceleration structures it is referred to chapter 2.

Rendering of shadows is a well researched topic and therefore only a brief overview of recent and relevant work is given here. For further information it is referred to [Eisemann et al., 2011] and [Hasenfratz et al., 2003].

Many methods exist for the task of rendering shadows. Starting with the work by [Williams, 1978] there have been many approaches to image-based methods. There, the occluding objects are stored in a so called shadow map first. In a second pass it is possible to determine with only one texture lookup of the shadow map which objects are visible from the light source. Lighting therefore has to be computed for exactly these objects, while all non-visible objects from the light source have to be shaded. This standard process of shadow mapping is fast but tends to have visible aliasing artifacts

if the resolution of the shadow map is not big enough. In this form, it is only possible to produce hard shadows, where the light source has no volume at all but is only represented by a single point in space.

Percentage closer filtering [Reeves et al., 1987] is one possibility to reduce the problem with aliasing through blurring of the shadow edges. It works by taking not only one but multiple nearby texture lookups of the shadow map and using this to calculate the percentage of visibility from the light source. With adjustments to this it is possible to approximate soft shadows from area light sources [Fernando, 2005]. There, the size of the filter kernel is adjusted according to the distance of the occluder. With this approximation the shadow is not physically accurate but the results are sufficient in many cases.

Other concepts to create shadows are geometry-based methods with the generation of shadow volumes that enclose the shadowed space [Crow, 1977]. It is possible to create correct shadows for point lights with hard shadow edges but it also needs some adjustments to create soft shadows with this idea [Assarsson and Akenine-Möller, 2003] [Laine et al., 2005]. In general, image-based methods using some kind of shadow mapping algorithm are more popular in comparison to geometry-based methods. This is due to performance reasons and a greater versatility and applicability of shadow mapping algorithms, but both approaches benefit from rasterization and are therefore fast.

A different approach to compute shadows is usually done with some kind of ray tracing algorithm. For each point that has to be tested for lighting a ray is constructed starting in that point and ending in the light source. If the ray is not intersecting scene geometry on this path, then the point is lit, otherwise it is shadowed. This approach is more versatile in comparison to the previous ideas but as explained in chapter 2 the calculation of the intersection between rays and scene geometry is rather slow.

5.3 Efficient Line Space Information

The goal is to compute ray traced shadows without testing the scene geometry for intersection. For this, a LS data structure is created based on the scene geometry, which however does not need the scene information afterwards. In that, the data structure is similar to sparse voxel octrees as they are for example used in voxel cone tracing [Crassin et al., 2011]. In contrast to octress, the leaf-nodes do not store contained scene geometry at all. Instead a LS is stored in every subdivided (= non-leaf) node

and the deepest level of the tree is ignored, as proposed previously in chapter 4. This therefore leads to approximated shadows comparable to those of sparse voxel octrees [Laine and Karras, 2011] [Kämpe et al., 2013] [Sintorn et al., 2014]. By using the directional data of the LS and its early termination criterion, it is possible to accelerate the shadow computation even further. For a compact overview, the binary visibility information within the LS from subsection 4.2.2 is first recapped and afterwards revised to fit the requirements for efficient GPU storage.

5.3.1 Binary Visibility Information within Shafts

As before, the LS is used as aggregation of the directional visibility data per shaft, which only consist of the information, whether the shaft is empty or non-empty. This binary visibility information is applied to all rays that start and end in the boundary patches of the given shaft, and as before, it is symmetrical as the start and end patches can be switched and give the same visibility result. Consequently, the now presented work relies on the same data and construction algorithms as chapter 4, where masking is used in order to precompute which subnodes of the current node are intersected by a certain shaft, as presented in subsection 4.2.3. If all intersected subnodes are empty, so they do not contain any geometry of the scene geometry at all, it is safe to conclude that all possible rays within this shaft are unable to intersect scene geometry inside of this node and all potentially intersected subnodes. For this, a shaft only needs the information if all intersected subnodes are empty or if there is at least one non-empty subnode intersecting the shaft. This is expressed in one bit of information. The LS contains this information for all possible shafts of one node and is represented as a 2D array or texture where the first dimension stands for the start side and the second dimension for the end side of a shaft.

Concerning the application in shadow generation, the now presented procedure is an inversion of the previous method. While before, LS information was used in order to skip empty shafts during traversal, the information is now additionally used in order to terminate the traversal as soon as a non-empty shaft in a certain tree depth is found. However, this has consequences in terms of the used parameter set (i.e. the subdivision parameter N and the maximum tree depth d) for the application, as the main focus now is not only on the runtime performance but also on the reduction of approximation errors, as will be shown in subsection 5.4.1.

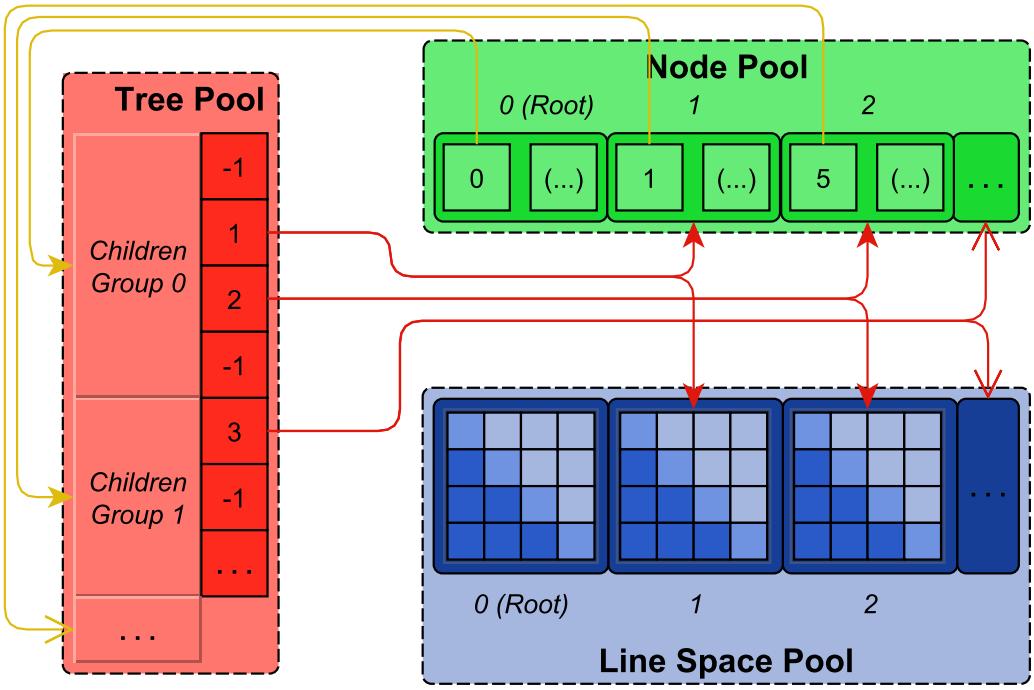


Figure 5.2: Management of the GPU data structure. The entries of the tree pool are clustered in groups of subnodes. Each entry of this pool is either set to a default value or refers to a subdivided node (stored in the node pool) and its corresponding LS (stored in the LS pool). Each node in the node pool contains the necessary data for its traversal and therefore a reference to its subnode group, which are traversed recursively.

5.3.2 GPU Data Structure

Adapting the idea of [Crassin et al., 2011], the data structure is implemented within data pools. Three different pools are used, which are stored in linear buffers on the GPU. In the first data pool (the tree pool) the tree information of the N-tree is stored, where all node relations are ordered in groups of subnodes. In the second data pool (the node pool) the information for all subdivided nodes is stored. This information is used to compute the traversal of subnodes of one node. The third data pool (the LS pool) is used for all LS information of the subdivided nodes. This concept is illustrated in Figure 5.2. This approach is implemented with OpenGL Compute Shaders and therefore all storage units are optimized for this.

The data structure is used in a way to only rely on subdivided nodes for traversal. Leaf nodes containing scene geometry are not needed and therefore not stored within the data pools. The tree pool consists of all

possible pointers that are needed to represent the hierarchy. The order of the pointers is based on groups of subnodes of subdivided nodes. All subnodes of one node are clustered to one subnode group. They have a specific internal order, which is the same for all subnode groups and depend on the local position within the parent node. If a subnode is subdivided as well, then the pointer is set to the index of the subnodes within the node pool. If the subnode is not subdivided, then the pointer is set to a default value, indicating that the traversal can skip this node. Instead of using a default value it is also possible to use negative values with a special meaning. With this it is possible to also store references to geometry information if needed.

The node pool is also used for the traversal. During traversal of the LS leaf nodes are skipped completely. For efficient memory usage therefore only subdivided nodes are stored within the node pool. A node within the node pool consists of different attributes as illustrated in Figure 5.3, which are used for the traversal. The main attribute of a node is a reference pointer to its subnode group within the tree pool. Other information needed for the traversal are the position of this node in world space and its size. It is possible to store additional information of a node like its resolution for the case that nodes have a variable number of subnodes.

The data of the LS pool is used as termination criterion in the traversal. The single bit information whether a shaft is empty is stored in this pool. For better incorporation with GPU-memory, the information of multiple shafts is combined to an integer value. The partition of the pool correlates with the node pool, so the n-th node in the node pool is related to the n-th LS in the LS pool. With this a pointer of the tree pool simultaneously refers to the corresponding node and its LS as shown in Figure 5.2. While a LS, as explained above and shown in the figure, is illustrated as a 2-dimensional data set, it is in fact implemented as 1-dimensional data within the buffer. It consists of a sequence of combined integer values and is therefore stored in an efficient way.

5.4 Shadow Computation

Using the LS, a data structure is used, that is able to decide whether a ray is probably going to intersect scene geometry through a given shaft or if it is definitely not going to intersect anything. Visibility of a light source in this context is therefore merely an approximation for the pure possibility of visibility. If a shaft only intersects empty subnodes, it is called empty itself. An empty shaft does not change the possibility of visibility and therefore the

Figure 5.3: Illustration of the structure of one *N*-tree node. It consists of different attributes, which are used for the traversal of the tree. All subdivided nodes have a reference to the subnode group index of this node. The reference is used in combination with this node's position and additional information like its resolution for the traversal of its subnodes. If it has no subdivided subnodes, then the reference is set to a default value, indicating that this node may be skipped during traversal.

light source counts as "visible". If a shaft intersects at least one non-empty subnode, then the shaft is called non-empty. A non-empty shaft may be able to block the visibility of a light source and as a result this shaft is classified as "occluding". Note that this is a rather conservative estimation of occlusion. A ray to a light source within a shaft may be declared as occluded even though it may pass by the scene geometry contained in the subnodes. An example to this is shown in Figure 5.4. Although this approximation may result in places that are wrongly occluded, this technique allows for a quick shadow traversal without the need to test the actual scene geometry for intersection. The benefit is not only that the test for occlusion is faster than the typical occlusion test in ray tracing, but furthermore, it is not necessary to store the scene geometry in the data structure at all. With this, the data structure is mostly scene-independent.

5.4.1 Shadow Ray Traversal with Approximations

Normally, the traversal for shadow computation traverses through the data structure until a node with scene geometry is found. This geometry is then tested for intersection. If an intersection is found within the node, then the traversal can end with a positive result. Otherwise it has to continue until an intersection is found or the data structures boundary is reached. Depending on the data structure the tests for intersection with scene geometry may be more costly than the traversal of the data structure itself. This fact is used in two ways. On the one hand theLS results in an implicit intersection test of the scene geometry, which is done through the shaft information that is precomputed during initialization of the LS, so no intersection tests with scene geometry need to be done during rendering. On the other hand the data structure traversal is accelerated by skipping the deepest level of

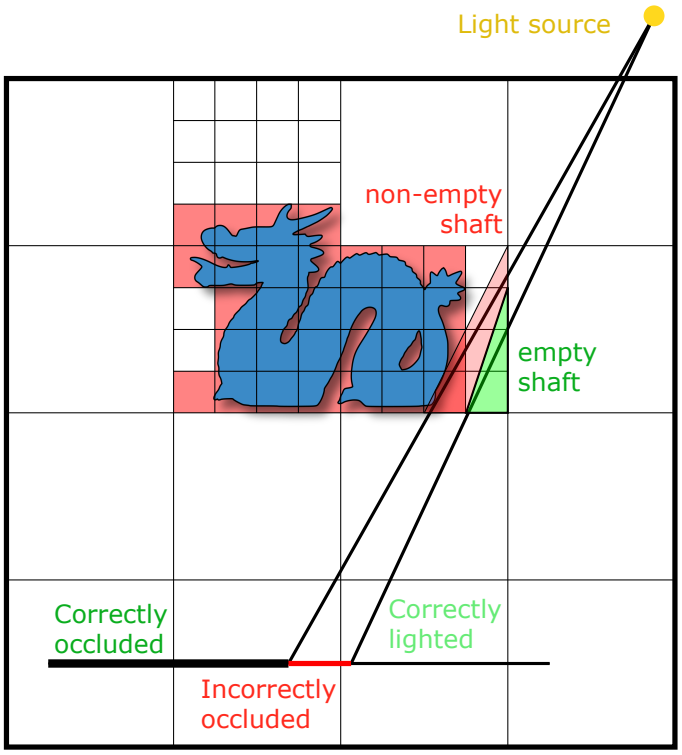


Figure 5.4: Illustration of the LS used for shadow computation. The object on the bottom is partially occluded by the dragon. While the occlusion on the left of the object is correct, the usage of the LS also results in incorrect occlusion. This inaccuracy is based on non-empty shafts in which the shadow rays would normally pass by the scene geometry without intersecting it, but are wrongly classified as occluded. Note that this inaccuracy is greatly reduced by using a higher subdivision parameter for the underlying N-tree.

the hierarchy and using the shafts as an approximation of the underlying deepest level. This additionally changes the optimal parameter sets. While in chapter 4 the main goal was to reduce the total number of intersection tests with scene geometry in order to accelerate the traversal, this is now not necessary anymore and it is more important to reduce the number of nodes that are traversed. Consequently, the subdivision parameter N and the maximum tree depth d are different and may be smaller in comparison to the previous approach.

The final traversal algorithm is shown in Algorithm 5. All nodes in the data structure are either subdivided and contain a LS or are leaf nodes and contain scene geometry. The latter are not needed during traversal as explained above, so this is checked first. If the node is subdivided, then the entry for the ray within the LS of this node is checked (line 2). If the entry is

Algorithm 5 The recursive LS traversal algorithm for shadow computation.

```
1:  procedure TRAVERSE(Ray r, Node n)
2:      if N has subnodes and LS(r, N) is true then
3:          if tree depth d is reached then
4:              return true
5:          else
6:              while subnodes left do
7:                  s ← next subnode in direction of r
8:                  if s is non-empty then
9:                      if TRAVERSE(r, s) then
10:                         return true
11:                     end if
12:                 end if
13:             end while
14:         end if
15:     end if
16:     return false
17: end procedure
```

not set, it means that the according shaft of this ray is empty and therefore, the ray is not able to hit anything within this node. Further inspection of this node can be skipped. If the entry is set, so the according shaft is non-empty, then the traversal of this node continues. Next it is tested whether the current node is deep enough in the tree hierarchy (line 3). A node is called deep enough, if it is on the second to last level in the hierarchy, so all subnodes of this node are leaf nodes. With this it may be possible that scene geometry is intersected by the ray and the ray gets accepted as occluded. Note that this is the part where the shadow is approximated. If the current recursion depth is not yet deep enough, then the subnodes intersecting the ray are being recursively tested for intersection (line 9).

A drawback using the LS without scene geometry is that it does not work if the ray starts within a shaft. The information stored within this shaft can not be applied to the ray because of the uncertainty how the objects within the shaft may be positioned in relation to the ray. Therefore, the node where the ray starts has to be skipped in the process and the traversal needs to start with the next node. This leads to a loss in quality in detailed areas.

Figure 5.5: Rendering results for the Dragon Scene using a single shadow ray to the light source's center point. The shadows in the left image are computed with the LS with a low-valued parameter set for illustration (a subdivision parameter N of 5 and tree depth d of 3) while the shadows in the right images are computed with a BVH as ground truth data. Using the low values in the LS parameter set leads to a shadow silhouette with visible errors in the case that a single shadow ray is used (bright areas in the heatmap).

5.4.2 Soft Shadow Computation

Using the LS as termination criterion for the traversal has the benefits of a faster occlusion test and it may need less memory space. The downside to this is the loss in accuracy, which comes from the approximation of the scene geometry with shafts as explained above. This effect is observable at the edges of the shadow regions where in general the LS produces more occluded areas as other approaches (see Figure 5.5). Then again, in most cases there are no point lights required but area lights, which do normally not produce hard shadow edges but soft transitions between occluded and non-occluded areas. Most approaches try to generate this effect by using multiple samples of the area light and calculating the percentage of non-occluded samples for lighting. By using this technique combined with the LS, the approximative nature of LS generated shadows become negligible (see Figure 5.7).

Though the shadows generated by this are overly shadowed, the difference is almost not visible, especially when many samples are used. In addition the inaccuracy caused by the LS is especially less significant the bigger the area light sources become. This is due to the fact that less samples near the geometry edge are falsely occluded by the LS. An example of

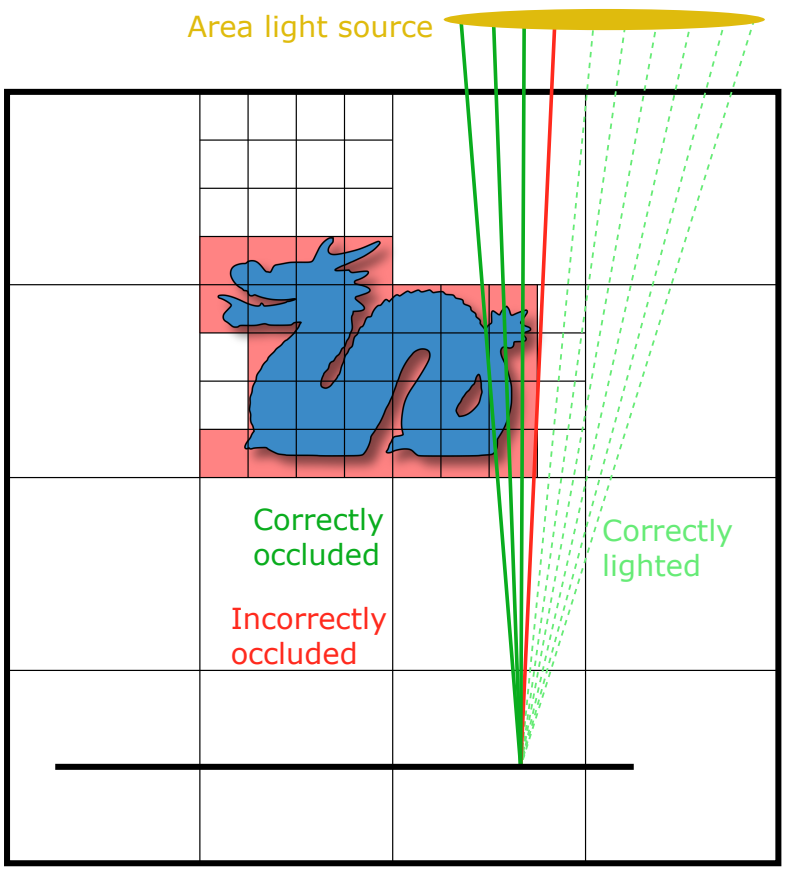


Figure 5.6: Illustration of the LS used for soft shadow computation. In this example 10 samples of the light source are used to calculate occlusion, while one of those samples is wrongly occluded. Note that as with normal shadows the accuracy of soft shadows based on the LS is greatly reduced by using a higher subdivision parameter for the underlying N-tree.

soft shadows generated by the LS is shown in Figure 5.6. By only testing the visibility based on shafts and not on the actual scene geometry, the traversal of the shadow rays is also more coherent and therefore better suitable for parallelization with the GPU.

5.5 Results

The approach is implemented on a NVidia GeForce GTX 1080 system with an Intel i7-6800k 3.6 GHz CPU and used GLSL Compute Shaders for GPU computing. As test scenes typical test models are used, that have different numbers of triangles and different characteristics (Bunny 69k

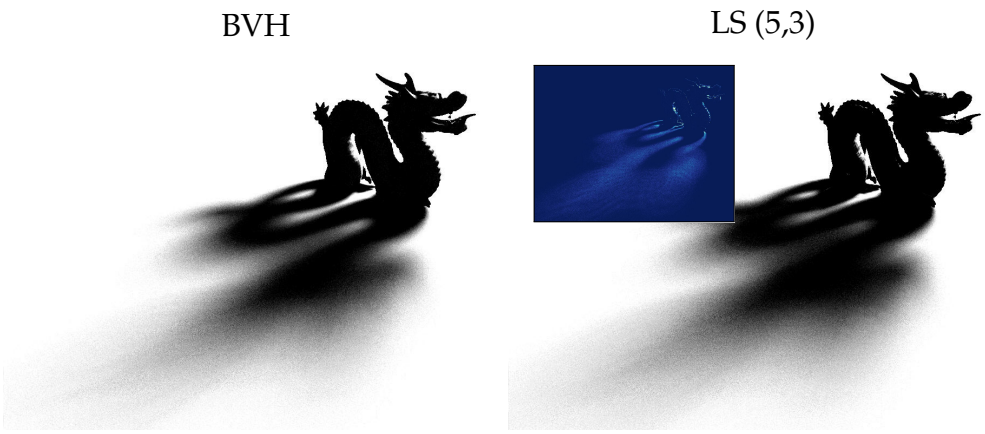


Figure 5.7: Rendering results for the Dragon Scene using 25 sampled shadow rays to an area light. As before a low-valued LS parameter set and a BVH as ground truth data are used. In soft shadow computation image errors produced by the LS (bright areas in the heatmap) are nearly not visible.

triangles, Dragon 871k triangles, Buddha 1087k triangles and Dragon and Buddha combined) on top of a simple plane. All scenes were rendered with a resolution of 1024×1024. While the geometry is rendered using typical forward rendering first, the shadows are applied using one of the techniques mentioned. Different camera angles are used and the average run time is taken as result. The shown method is compared with state-of-the-art BVH accelerated ray tracing using the GPU as proposed by Aila and Laine [Aila and Laine, 2009].

The size of the LS and the build time vary significantly with the subdivision parameter N and the maximum tree depth d used for the underlying N-tree. In contrast to chapter 4 these parameters may be smaller, as the main focus now is not on reduction of necessary intersection tests with scene geometry but reduction of traversed nodes. Therefore, a subdivision parameter between 4 and 8 is used here with a maximum recursive tree depth of either 3 or 4. The LS is constructed on the GPU on top of the previously build N-tree. As with most other complex ray tracing data structures, interactive initialization timings are not achieved. The number of nodes containing a LS and the resulting size of the data structure with different parameter sets are given in Table 5.1. There, the build timings for the LS on top of the N-tree and the rendering timings are shown as well.

In comparison to BVH-based ray tracing, the LS does not need any intersection tests with scene geometry. With this, it has a substantially better performance in computing soft shadows, as presented in Figure 5.8.

Scene		BVH	Line Space (N, d) (4, 4)	(5, 4)	(6, 3)	(7, 3)	(8, 3)
BUNNY (69k tris)	**Size (MB)**	4.3	9.4	92	12.3	45.6	115.2
	#k Nodes	31.4	13.0	57.2	3.9	8.1	12.4
	Init (ms)	–	28	489	100	547	1779
	Perf (FPS)	95.3	58.7	52.5	69.9	62.2	59.6
	Error (RMSE)	–	.022	.013	.019	.014	.009
DRAGON (871k tris)	**Size (MB)**	35.0	9.9	97.7	13.1	48.7	122.2
	#k Nodes	82.9	13.8	60.7	4.1	8.7	13.2
	Init (ms)	–	34	533	105	558	1788
	Perf (FPS)	22.4	23.7	21.1	33.6	32.2	29.8
	Error (RMSE)	–	.041	.031	.038	.028	.021
BUDDHA (1087k tris)	**Size (MB)**	41.6	9.4	67.1	12.1	35.4	95.7
	#k Nodes	69.9	13.1	41.7	3.8	6.3	10.3
	Init (ms)	–	30	355	98	397	1432
	Perf (FPS)	16.4	47.5	42.6	60.8	57.3	19.7
	Error (RMSE)	–	.042	.034	.041	.035	.027
BUDDHA & DRAGON (1959k tris)	**Size (MB)**	76.6	12.7	89.4	15.8	46.2	127.6
	#k Nodes	152.7	17.7	55.6	5.0	8.2	13.7
	Init (ms)	–	42	475	131	523	1928
	Perf (FPS)	12.7	23.6	22.4	26.6	24.9	24.3
	Error (RMSE)	–	.044	.035	.044	.037	.029

Table 5.1: Comparison of the size in MB, number of subdivided nodes (in 1000), build time in ms, rendering times in frames per second and image errors in root-mean-square-error for different parameter sets of the LS and BVH based on the work by Aila and Laine [Aila and Laine, 2009]. Every parameter set of the LS is given as (N, d), where N stands for the subdivision parameter and d for the maximum tree depth of the used N-tree. For the rendering the time to compute soft shadows of one area light source with 25 shadow rays are measured with an image resolution of 1024×1024. It is visible that the LS has better rendering performance in medium and big scenes, but not in scenes with only a small number of triangles.

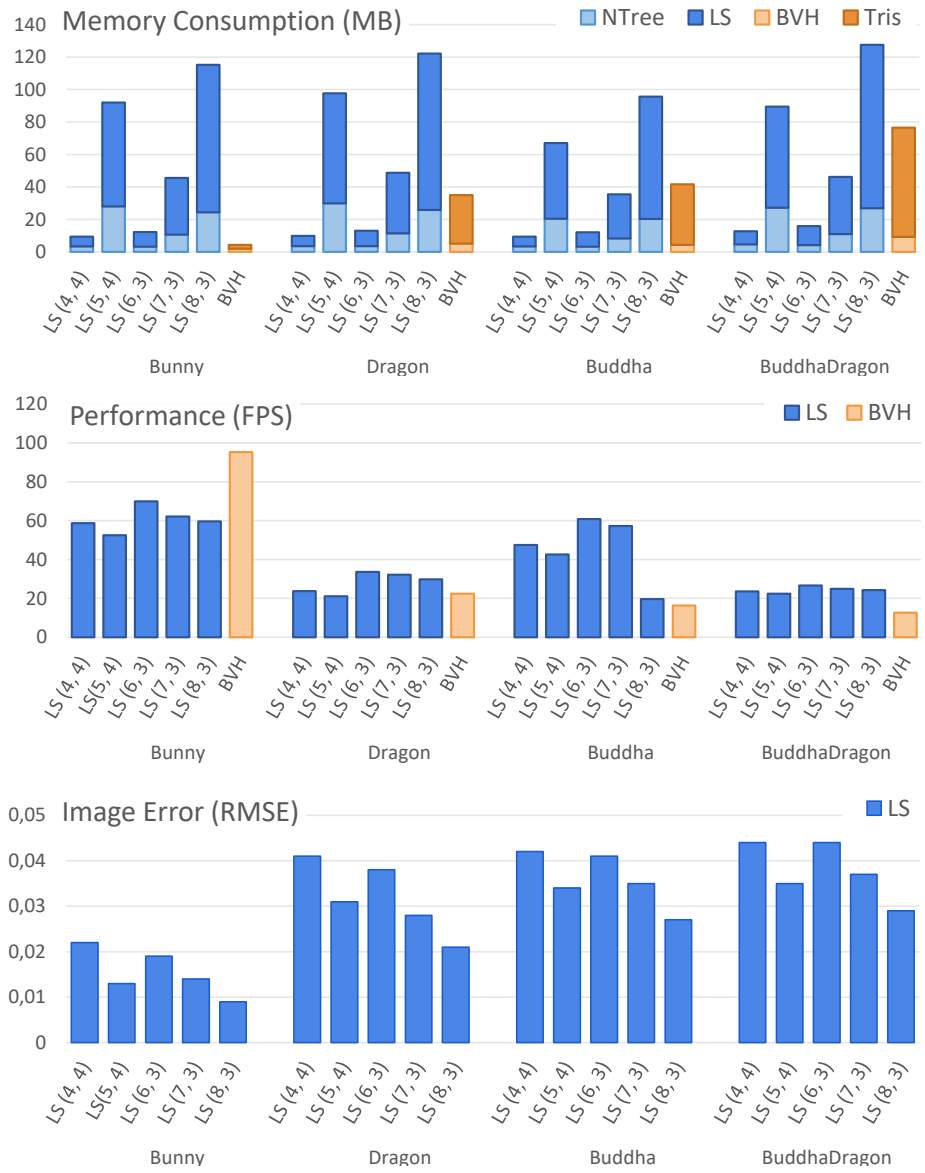


Figure 5.8: Overview of memory usage (in MB), performance (in FPS) and image errors (in RMSE) of the LS using different parameter sets and BVH for the test scenes. Using high values of N or d lead to high memory consumption but less image errors, as shown by the parameter sets of LS (5, 4) and LS (8, 3). The used memory size is nearly scene independent and mainly depends on the used parameters. The BVH size is rather small but in contrast to the LS it additionally needs the triangle information of the scene. In general, the LS results in higher performance in more complex scenes with a high number of scene triangles. Data from Table 5.1.

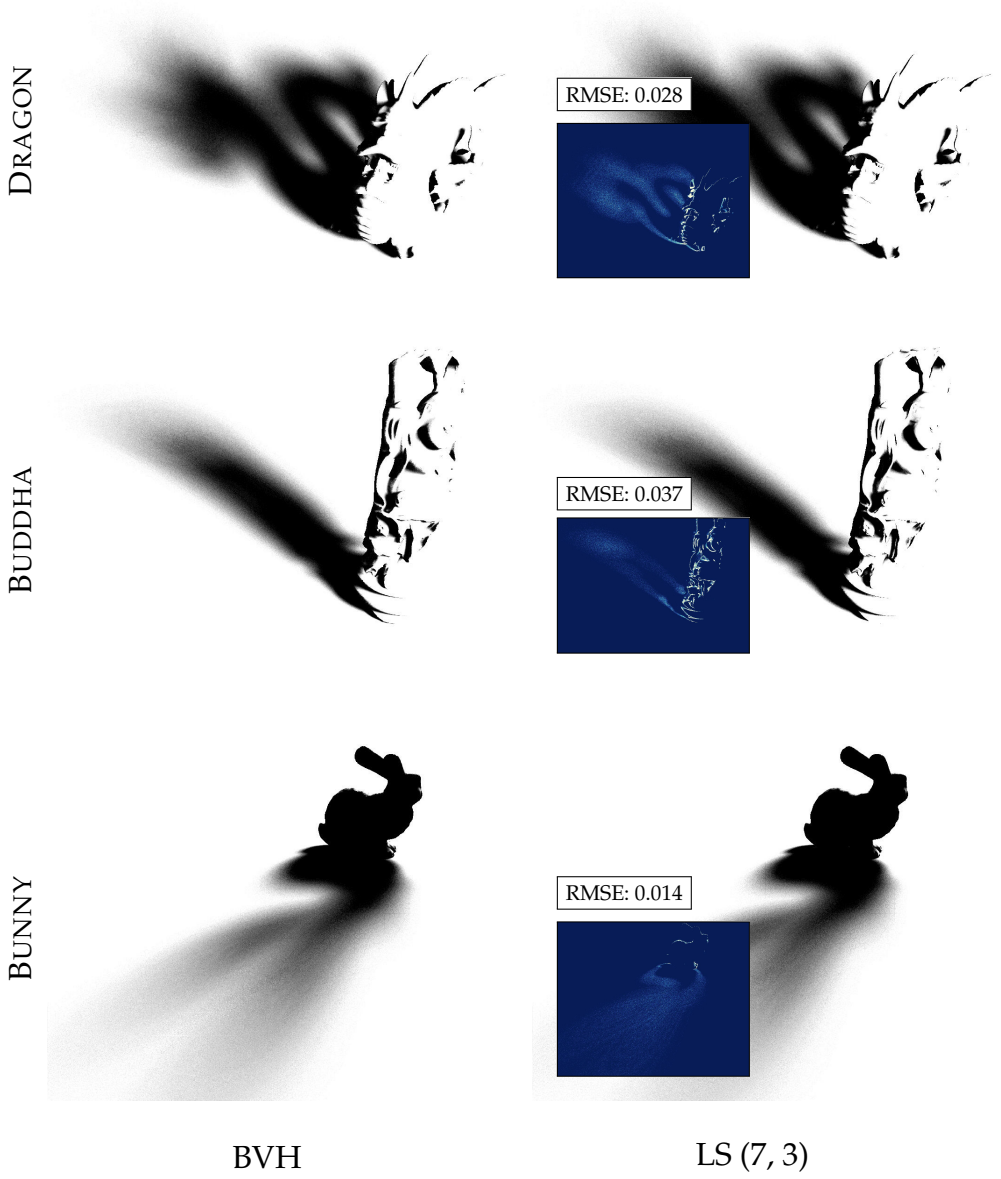


Figure 5.9: Comparison to a BVH-based ray tracer. The LS inaccuracies (bright areas in the heatmaps) mentioned above are only barely noticeable, even with smaller parameter sets. The produced shadows are slightly darker and the skipping of the first node in the LS traversal leads to less occlusion in detailed areas. Overall, the results are nearly equivalent to ground truth renderings, but have better performance and do not need to store the scene geometry, which grants less memory usage and different opportunities in the future.

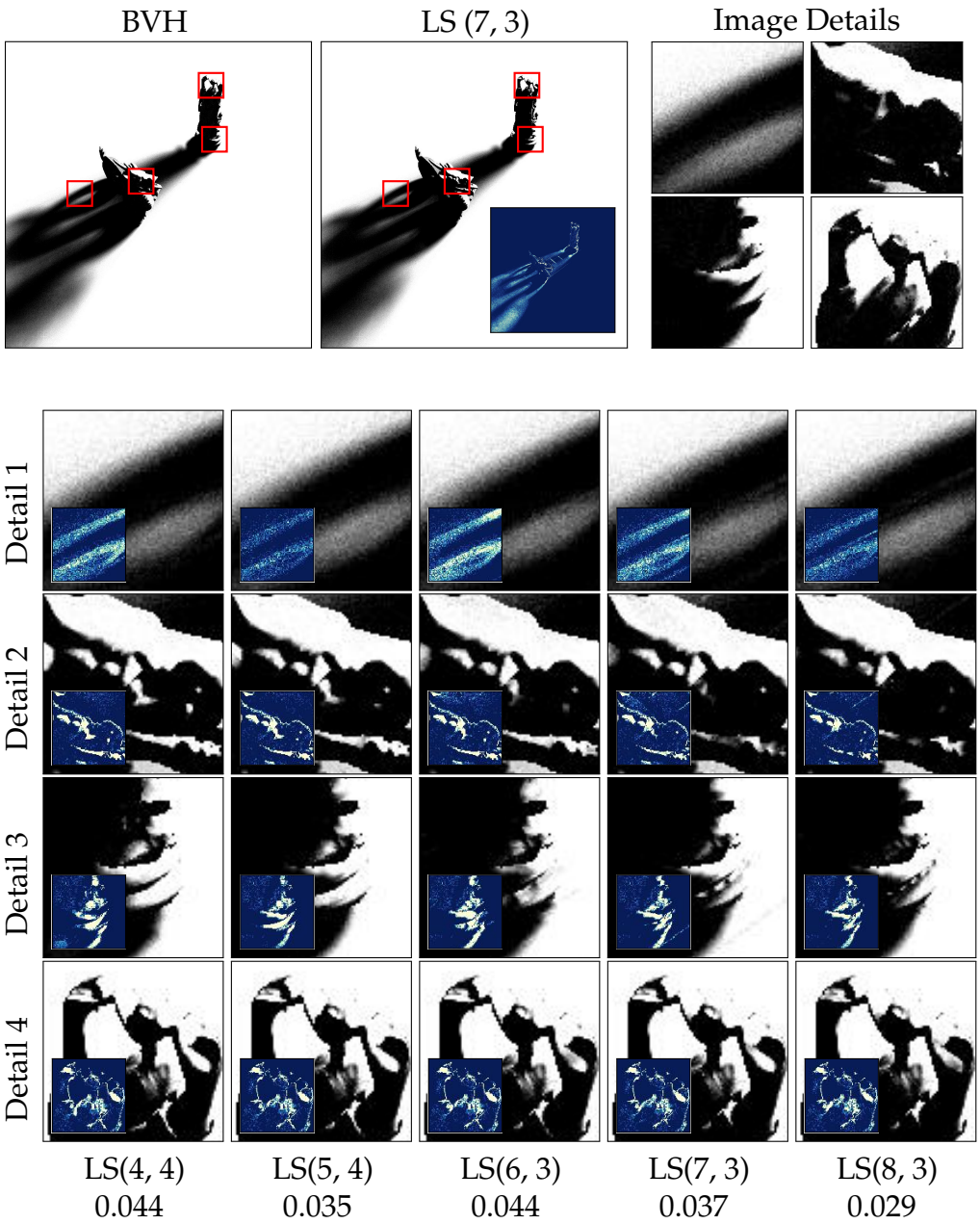


Figure 5.10: Detailed test results of the presented technique (used scene Buddha + Dragon, 1959k triangles). Soft shadows were generated using one area light source with 25 samples and compared to ground truth. RMSE error values are presented (bright areas in the heatmaps) along with the parameter set (as LS(N, d)). Magnified images with varying LS parameter sets are illustrated to demonstrate the differences caused by the early ray termination of the LS. Parameter sets with higher values lead to more accurate results but much higher memory consumption, as shown in Table 5.1.

The results may differ significantly in scenes that do not fit in the GPU memory, but this needs to be further investigated and is beyond the scope of this work. It is visible that the rendering performance of the LS in comparison to the BVH is better in medium and big scenes with up to more than $3\times$ the performance, whereas the BVH achieves faster results in the small scene. Overall, it is visible that the quality of the BVH in terms of performance is mainly influenced by the number of triangles used in the scene. In contrast, the performance results of the LS are as expected more stable with varying numbers of scene triangles, but are more influenced by the spatial structure of the scene. This is due to the fact that the LS does not store the scene geometry in any kind, but an approximation of the scene within the abstraction of the shafts.

In terms of memory consumption the BVH is mainly affected by the number of scene triangles, which have to be stored in addition to the node information of the hierarchy. In contrast the LS does not need to store any triangle information at all and it only relies on the node and LS data. Figure 5.8 shows the memory consumption for the mentioned scenes for the BVH and the LS with the tested parameter sets. The size of the LS significantly depends on the used parameter set, where a higher value of the subdivision parameter N or the maximum tree depth d leads to a much higher memory consumption. As with the rendering performance, it is visible that the memory size of the LS is nearly scene independent and quite stable over all used scenes. With this it is possible to give an early estimation of the memory size for a given parameter set before actually building it.

As explained before, the usage of the LS for shadow generation is flawed with approximations due to the abstraction with the shafts. This is especially visible when only one shadow ray is used per pixel. In soft shadow computation, this problem is only barely noticeable and is reduced by increasing the parameter set values used, as shown in Figure 5.7.

The rendering results for soft shadow computation are shown in Figure 5.10 and Figure 5.9. The usage of the LS leads to faster results in bigger scenes with only a minimal loss in accuracy. Nevertheless, the shadows are only approximated as explained above and so the quality of the LS results is not as precise compared to accurate computations using BVH accelerated ray tracing. The results show the mentioned difficulties of the approach. It is observable that different parameter sets of the LS lead to different results in the shadow generation, which is explained with the varying orientation and size of the shafts within nodes with different resolutions. Furthermore, it is visible, that shadows in detailed areas get lost with the LS and the shadows are in total slightly darker. But overall, these inaccuracies are only

barely noticeable. The results are nearly similar to the precise results of the BVH traced method and are therefore quite acceptable.

5.6 Conclusion and Future Work

A novel approach in calculating fast shadows with the GPU is presented. Using the LS as a data structure for precomputed approximated occlusion values leads to a fast traversal of shadow rays. Though the produced shadows are not absolutely accurate, they are precise enough for soft shadows. Moreover, it is shown that the results are faster in production than typical ray tracing methods, with in some cases more than $3\times$ the performance of a state-of-the-art BVH. Furthermore, the data structure does not need information about the scene geometry for shadow calculation and it is therefore able to have a smaller memory consumption.

While the data structure is optimized for SIMD parallel usage on modern GPUs, the initialization is still computed on the CPU. As future work, it is therefore beneficial to accelerate the initialization process by computing it in parallel on the GPU. With that it may be possible to work with dynamic scenes and the LS may then become an alternative for typical soft shadow techniques for dynamic scenes. Apart from this, it is possible to investigate the impact of storing not only binary information for shafts. It may be possible to precompute ambient occlusion values per shaft and store them within the data structure. This would accelerate the traversal step further and may therefore lead to better results. Another option would be to not only store binary visibility information in a shaft, but to use a counter for the number of objects intersecting the shaft. This may give the possibility for dynamic updates, so that it is not necessary to recompute the full LS once an object is moving.

Furthermore, the extents of the LS usefulness need to be examined. In chapter 4 it was shown that it is possible to speed up the general traversal in ray tracing. Additionally it is possible to use the LS visibility information for shadow computations. The next step is to examine, whether other information for shafts also grant the possibility to compute indirect or global illumination. This is done in chapter 6.

While the LS is for now computed for the complete scene, an obvious step for future work is to use it based on single objects, rather than the scene. This may bring some advantages like object instancing and rigid body transformations without changing the LS data. Moreover this can be beneficial for initialization time and memory consumption of the data structure as well as reduce the resulting errors in the final images. Further

research in this topic is done in chapter 7 and again in application with soft shadow and ambient occlusion calculation in chapter 8.

CHAPTER 6

APPROXIMATION OF INDIRECT LIGHTING WITH REPRESENTATIVE SHAFT CANDIDATES

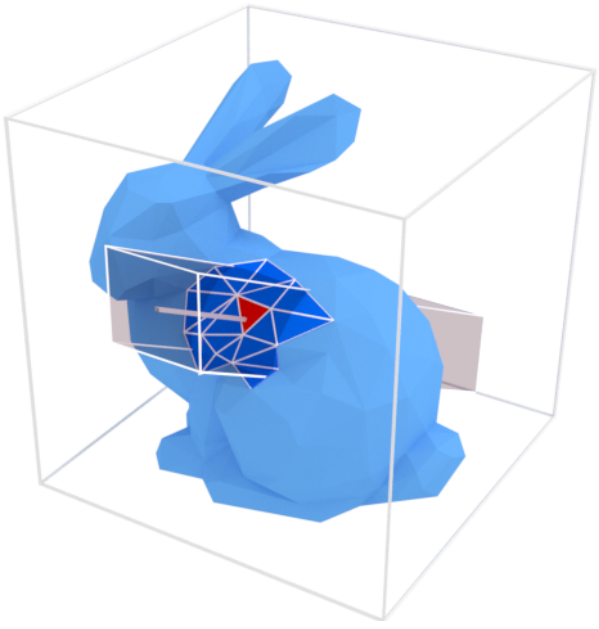


Figure 6.1: The representative candidate triangle (shown in red) is the triangle that is used to approximate the geometry in a given shaft during runtime. It is found by intersection computation between the geometry and the centroid ray of the shaft.

The contents of this chapter are based on the publication:

> Keul, K., Koß, T., and Müller, S. (2018). Fast indirect lighting approximations using the representative candidate line space. *Journal of International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG)*, 26:11–20

6.1 Introduction

With the precomputed binary visibility information of the LS, it was possible to achieve significant improvements in ray traversal performance. However, until now this was limited to trivial rendering effects. In chapter 4 it was demonstrated, how the binary classification of shafts in terms of emptiness was used for an effective empty space skipping technique. This was extended in chapter 5 for approximation of soft shadows without further intersection tests during runtime. With this, the LS achieved a new step in the big vision of ray traversal computation with only a single request based on precomputed visibility information. This was accomplished by integration into typical spatial data structures that were used superficially to remain the necessary structure. Apart from that, the LS enabled significant improvement in runtime performance due to the fast blocker approximation without the need of intersection tests. The next step, which is presented in this chapter, is to extend the contained shaft information of the LS to non-binary visibility information in order to facilitate fast traversal of indirect lighting and global illumination rays.

Calculation of global illumination and indirect lighting is a non-trivial task, which significantly improves realism of generated images and renders the possibility for photo realistic effects. The two main ways for computation of global illumination are depth-based rasterization techniques and ray tracing. The former is typically used in interactive and real-time rendering due to the high performance that is achieved. The basic idea is to determine the visible scene primitives through projection to the screen in object order. Adding complex rendering effects like global illumination without ray tracing is a non-trivial task and suffers from different quality problems [Ritschel et al., 2012]. In ray tracing the visible surfaces are calculated per screen pixel. As presented in chapter 2, by using additional rays per pixel it is possible to calculate complex rendering effects and indirect lighting.

In this chapter, the LS is further extended by precomputing a representative candidate (i.e. a triangle) for each non-empty shaft. This leads to significantly faster but approximated results, which can be used for the

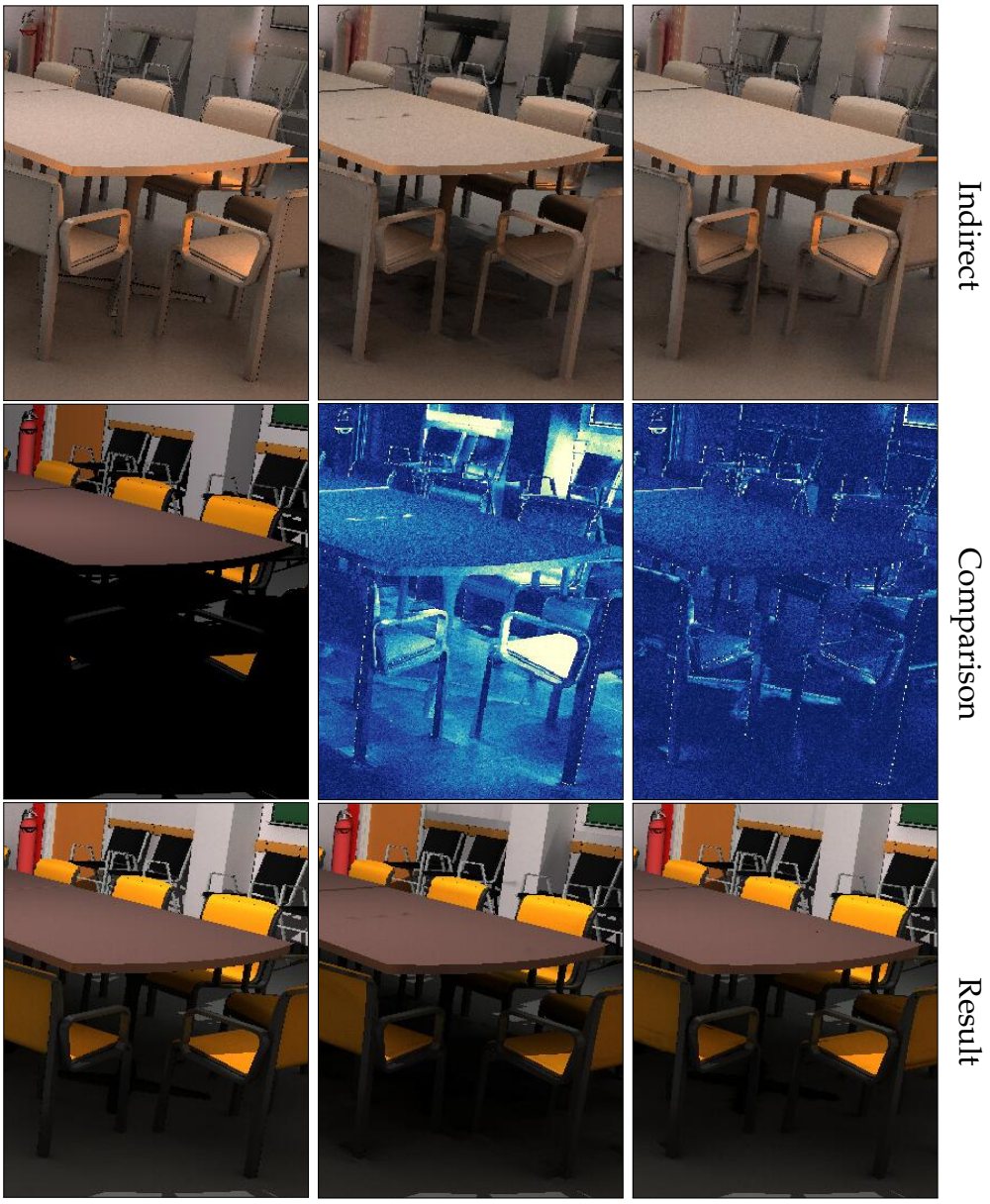


Figure 6.2: Example of the presented technique. The left column shows correct results as reference, the other columns show the utilization of precomputed scene primitives in the LS using a low and a high integration depth parameter. In the top row indirect illumination is presented. The middle row shows a comparison to ground truth, where the left image presents only direct lighting and the middle and right images show heatmaps with bright areas marking high error values for indirect illumination. The last row consists of the final images.

acceleration of indirect lighting computations. While the results suffer from approximation artifacts, it was shown in [Yu et al., 2009] that indirect illumination does not require correct results and therefore the artifacts can be disregarded in this context, as illustrated in Figure 6.2. Moreover, a general LS description on basis of bounding boxes is used. With this the presented approach can be applied to almost every spatial data structure used as base structure. This applicability is demonstrated with the N-tree, the regular recursive grid structure which was used before, and BVHs, which are the most used acceleration structure at the moment. Therefore the general utility in terms of accelerating performance is shown.

As the rendering of indirect lighting and global illumination is a well studied and complex topic, it is referred to [Wald et al., 2003b] [Pharr et al., 2016] and [Ritschel et al., 2012] for a broad overview of techniques and possibilities in ray tracing and rasterization-based techniques. The focus of this chapter is specifically on the acceleration of approximated intersection point computation through the usage of sophisticated data structures. For further information on typically used acceleration structures in ray tracing, it is referred to chapter 2. The main contributions here are:

- An approach for precomputation of possible intersection candidates based on the simplification of clustering rays into shafts in the LS with the application of indirect lighting calculation without the need of intersection tests.
- A generalization of the LS to voxels and bounding boxes and therefore the adaption to almost all commonly used spatial data structures as base structure.
- An extension to the previous efficient structure based on data pools for storing and using the more complex non-binary visibility information on the GPU.
- An evaluation in terms of performance, memory consumption, initialization speed and quality of the base data structure in combination with the LS and the comparison to the pure data structure.

6.2 Line Space with Representative Candidate Data

In order to present the main idea of the representative candidate LS, the necessary LS basics are shortly revised first. As proposed before in chapter 4,

Algorithm 6 The accelerated intersection algorithm between a ray and a LS node. This is used when the integration depth d is reached during traversal.

```
procedure INTERSECT(Ray r, Node n)
    (t_start, t_end) ← points where ray r intersects box of node n
    i ← CALCPATCH(t_start)                                  ▷ start patch
    j ← CALCPATCH(t_end)                                    ▷ end patch
    shaftID ← CALCSHAFT(i, j)
    triangle t ← GETCANDIDATETRIANGLE(shaftID)
    if triangle t exists then
        return intersect ray r with triangle t
    end if
    return ∅
end procedure
```

it is a data structure providing directional information for a given voxel or bounding box. The six faces of the box are equally divided into N^2 rectangular patches. Pairs of those patches that are arranged on different box faces are defined as *shafts*. The volume of a shaft is the convex set of all line segments connecting any point in the start patch with any point in the end patch. The LS stores arbitrary data for each shaft. A LS where a substitution of the start and end patches leads to the same result as the original one is called *symmetric*. There are $30N^4$ shafts in a non-symmetric LS and $15N^4$ shafts in a symmetric LS, therefore potentially resulting in a big memory consumption, as shown before.

6.2.1 Representative Candidate per Shaft

Until now, the LS was only used to store binary visibility information, i.e. whether a given shaft contains any geometry at all. This approach was utilized for empty space skipping in chapter 4 and accelerated shadow computation in chapter 5. Here it is extended by storing a representative triangle for each shaft, which serves as an approximation for the geometry inside of the shaft. The stored information can be used during the traversal step of ray tracing to get a possible intersection between a given ray and the scene geometry. Instead of testing all candidate triangles of the given bounding box for intersections, only the previously stored representative triangle is considered. This intersection is approximated based on the ray and a node at the LS integration depth d, as presented in Algorithm 6.

The representative candidate LS is non-symmetric and therefore a shaft that starts in patch s and ends in patch e is seen as a different shaft than

the one starting in patch e and ending in patch s. The candidate used as shaft representative is the triangle that best approximates the object surface within the shaft. To find it, an intersection between the geometry and the ray defined by the centroids of the shaft's start and end patches is searched, as illustrated in Figure 6.1. If no intersection is found, the shaft is marked as empty. The percentage of empty LS shafts is typically between $30\% - 70\%$ for manifold meshes, and therefore, much memory is saved with an appropriate memory layout, as explained later.

The surface inside a shaft can be classified into three categories, as illustrated in Figure 6.3:

1. The surface is closed and covers the whole shaft width.
2. The shaft lies at the boundary of a surface or contains multiple disconnected surfaces.
3. The shaft is empty.

Since only the single triangle per shaft is stored, it is not inherently possible to distinguish the first two cases. The candidate triangle does not necessarily cover the whole shaft surface, as it is shown in Figure 6.1. To compensate this, the triangle is treated as infinite plane defined by its vertices. This approximation is used when searching for an intersection between a ray and the geometry contained in the shaft. Per-vertex normals can be interpolated by using the intersection parameters of the constructed plane. If the intersection point is outside of the triangle, it is computed by the extrapolation, therefore providing smooth normals for the whole shaft width. This is a rather big approximation, especially for highly curved surfaces, however, it lowers discontinuity artifacts. To reduce artifacts for shafts that are not fully covered by a surface, edges can be found by calculating the angle between the extrapolated normal and the mean normal of the triangle. If the angle is bigger than a given threshold, then the intersection is discarded.

6.2.2 Memory Layout for Non-Binary Shaft Information

The representative candidate LS stores a reference to a triangle for each shaft. In the presented case, this reference is a 32-bit index pointing to a buffer containing all triangles of the scene geometry. Depending on the storage layout of the scene geometry, more space efficient data types are possible. Since the LS stores data for every combination of start and end patch, it contains $M = 30N^4$ elements. While most of these shafts are empty

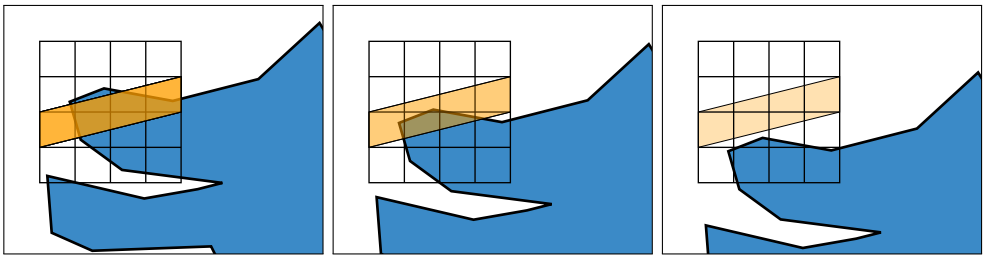


Figure 6.3: Shaft coverage by surface. **Left:** Shaft is completely covered by a single surface. **Middle:** Only partial coverage and multiple objects are present in coverage. **Right:** Shaft is empty and therefore completely uncovered.

and do not point to a valid triangle, it is possible to only store the shaft information of filled shafts. This is done by using a simple sparse scheme based on a bitset and offset buffer to skip empty shaft entries and only store filled shafts consecutively in memory. In addition, it is necessary to store one bit of information for each shaft, signaling whether the shaft is empty or filled. These bits are grouped in bitsets and are efficiently represented by 32-bit words. Moreover, an offset for every bitset is stored, which specifies where the corresponding filled shaft entries withing the shaft buffer are located. This is shown in Figure 6.4. The memory overhead of this scheme therefore sums up to $\frac{M}{16}$ 32-bit words. Finally, the sparse storage is more memory efficient compared to dense storage if less than $\frac{15}{16} \approx 93\%$ of the shafts are filled, which is always the case. The offset computation is done by a parallel prefix sum, efficiently calculated on the GPU. It should be noted that the bitsets are identical with the binary LS, and therefore, they are implicitly calculated. The data access with the sparse memory layout has a constant time complexity and is presented in algorithm Algorithm 7. 32-bit words are used for all bitset and offset operations due to the better interaction with GPU computations. However, this can be generalized for arbitrary sizes like 64-bit words.

6.3 General Line Space for Spatial Datastructures

Because of the construction on top of bounding boxes, the LS can be adapted and integrated into each spatial data structure that consists of bounding boxes in any way. It provides directional information in addition to the spatial subdivision and therefore is able to minimize the traversal cost.

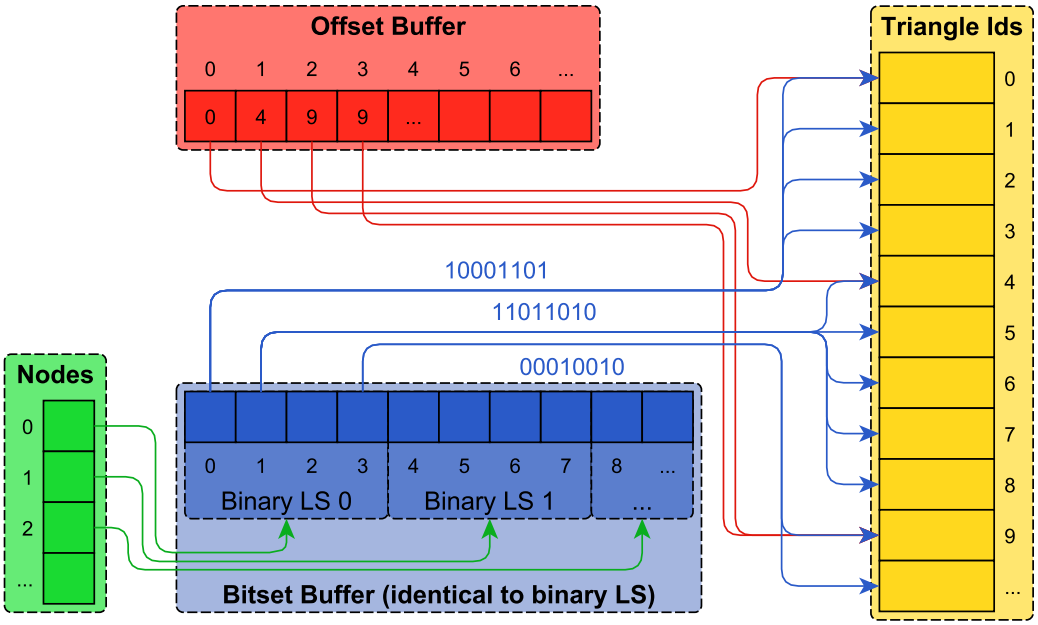


Figure 6.4: The sparse memory layout of the structure. Nodes have references to their bitsets (shown in green), which signal whether associated shafts are filled or empty. To access the triangle data, an additional offset per bitset is needed (shown in red).

Typical data structuces that can be used for this are Octrees, BVHs, k-d trees, uniform grids or recursive grids (such as the N-tree as shown before).

A representative candidate LS is generated for selected tree nodes of the data structure to approximate the scene geometry. While the produced errors are too striking for direct illumination of primary rays, they are less perceivable when used for indirect illumination, which is in accordance to [Yu et al., 2009], stating that accurate calculations are not required for global and indirect illumination. Therefore, the representative candidate is used as approximation instead of the correct triangle data to calculate the intersection points in indirect illumination. In terms of the underlying base data structure it is necessary to consider which nodes in the tree need to store the LS information.

6.3.1 Adaptation to *N*-tree

The data structure previously used for LS computations is the N-*tree*, which is a regular recursive grid that repeatedly subdivides the scene and the already produced subdivisions into N^3 equally sized bounding boxes. Here,

Algorithm 7 The algorithm presents the access of the shaft data in the sparse memory scheme.

```
procedure GETSPARSEDATA(ShaftIndex n)
    bitsetID ← ⌊n/32⌋
    bitset ← GETBITSET(bitsetID)
    bit ← n mod 32
    if (bitset & (1 ≪ bit)) ≠ 0 then
        offset ← GETOFFSET(bitsetID)
        id ← BITCOUNT(bitset ≪ (31 - bit)) - 1)
        return GETDATA(offset + id)
    else
        return ∅
    end if
end procedure
```

the size of these boxes is constrained to cubes. This simplifies some computations when building and traversing the N-tree while not having any detrimental effect for the data structure. The N-tree provides increased traversal performance in comparison to a regular Octree or single layer uniform grid. This is because the tree width of an N-tree with $N > 2$ can be larger than for Octrees, effectively lowering the tree depth. Since the bounding boxes of N-tree nodes are equally sized, the N-tree is a natural fit for LS computations, as shown before. The LS patch size for a specific tree depth is equal for all nodes and correlates with the size of the node subdivisions.

To use the N-tree with the representative candidate LS for indirect lighting approximations, first the N-tree is generated including the exact triangle data of the scene. This N-tree can be used for all exact computations like primary or shadow rays. Moreover, it is utilized for faster initialization of the representative candidates. Only the LS data on specific nodes in the N-tree is computed, which are determined by the integration depth d or a triangle count lower than T (i.e. $T = 8$). In the presented case the N-tree and also the representative candidate LS have parameter values $N = 6$ or $N = 10$ and $d = 2$ for LS utilization, which is consistent with the results shown before. The N value describes the subdivision parameter of the N-tree as well as the LS resolution. It was found to be accurate enough for the approximation while also granting sufficient performance. The integration depth parameter d also determines the performance, memory requirements and the approximation accuracy of the LS. Higher integration

depth values lead to more LS nodes and therefore higher computation times and memory consumption. However, lower integration depth values decrease the accuracy of scene approximations but significantly increase the traversal performance. This is due to the fact that the number of nodes greatly increases with the depth of the underlying tree. The shown value of $d = 2$ for the usage of LSs within the N-tree is sufficient for quality, traversal speed and memory consumption.

When traversing indirect rays, these LS nodes are treated as leaf nodes in the N-tree and are therefore able to terminate the traversal. Intersections with the scene are calculated by the procedure explained in subsection 6.2.1. The general traversal algorithm of the data structure does not need to be changed, only the handling of leaf nodes needs to be replaced by the appropriate LS calculations as shown in algorithm Algorithm 6.

6.3.2 Adaptation to BVH

By using a BVH, the scene is recursively subdivided by axis aligned bounding boxes that tightly enclose the geometry. Besides the N-tree, the BVH is also used as a base data structure for the LS in this work. The representative candidate in the LS nodes are used in the same way as described with the N-tree. Due to the reason that every BVH node branches into only 2 subnodes, the tree depth is normally much higher than for the N-tree. With this BVH nodes converge to the actual scene geometry and they are not constrained to be equal in size in every tree layer. Typically less nodes are needed in the BVH for scenes with a high amount of empty space. Again, the integration depth d and the triangle count T are used to determine whether a node is extended by a LS. Furthermore, it is possible to consider the box size as criterion for this determination, but this was not done here. The subdivision parameter of a LS node is independent from the base data structure and can be arbitrarily chosen. In this work it is set to a value of $N = 6$, as this value was shown to result in a good trade-off between performance and quality.

Nevertheless, the usage of a BVH with the representative candidate LS has two disadvantages. Since the bounding boxes are not cubical, the shaft patch size will slightly differ for each bounding box. The approximation artifacts in that case are not distributed in any predictive manner, and therefore have significant impact, depending on the used parameter set. This is visible in the results using low parameter sets for the BVH LS. Additionally, BVH nodes may overlap, especially in scenes where the triangle size is highly diverse. Therefore, multiple LS nodes and their shafts

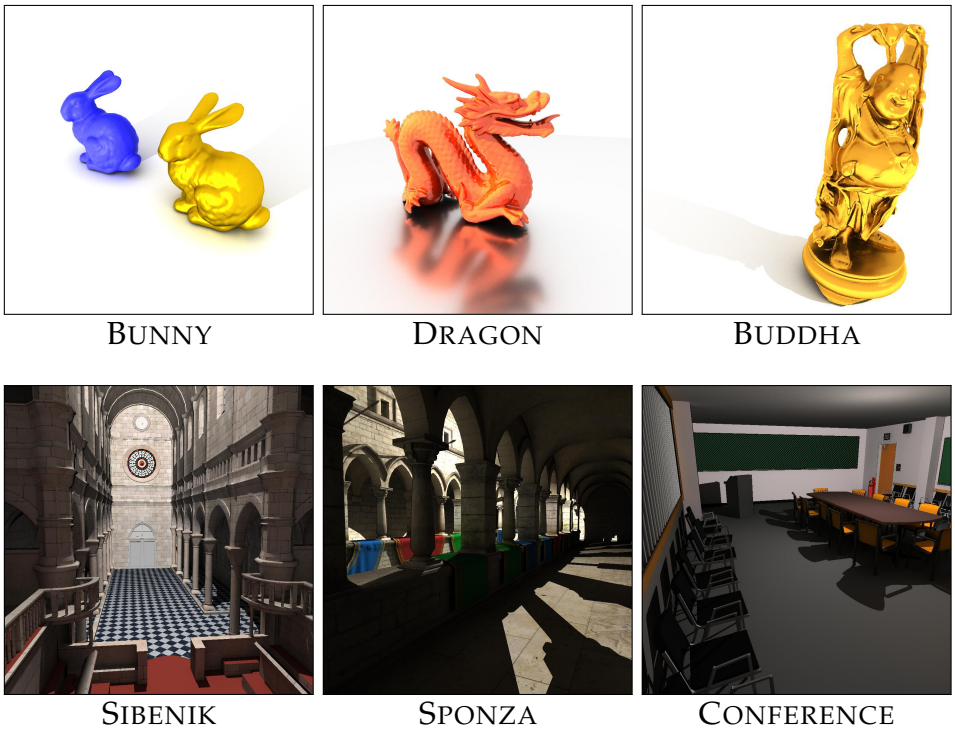


Figure 6.5: The evaluated test scenes. Renderings were done in 720p. Primary rays were calculated with rasterization, shadow rays were rendered with a binary LS and indirect rays were produced with the presented technique.

may overlap and approximate the same scene geometry using different representative candidates. These effects are mostly visible in architectural scenes, as shown in the results. A spatial split BVH construction might reduce these effects significantly, however, this was not used here.

6.4 Results

For the evaluation two different base data structures are used: the N-tree, a regular recursive grid as it was used before in chapter 4 and by [Billen and Dutré, 2016], and a state-of-the-art BVH algorithm [Aila et al., 2012], which is used as comparison in multiple related works. For the N-tree a subdivision parameter N of 6 and 10 and a hierarchical integration depth d of 3 is used, as those are the proposed parameters by chapter 4. No further optimizations of the BVH tree quality as recently proposed [Ganestam et al., 2015] [Yin and Li, 2014] are incorporated. The LS with precomputed representative shaft candidates is then used in a given

hierarchical tree depth of the base data structure. Within the N-tree this integration depth is set to the lowest branching nodes in the hierarchy, i.e. integration depth 2. The LS tree depth within the BVH is more versatile and can be set arbitrarily to achieve a good trade-off between performance and memory consumption as well as initialization time. The chosen integration depths and their impact are shown in the diagram and the visual results. As explained before, the LS subdivision parameter N within the BVH is set to a value of 6.

The quality and the differences in performance, initialization time and memory consumption are measured. For this purpose different widely used test scenes with special characteristics, as shown in Figure 6.5 are evaluated. In principle they are dividable into two categories: scenes containing a single object (BUNNY, DRAGON and BUDDHA) and architectural scenes (SIBENIK, SPONZA and CONFERENCE), which are more suitable for usage in video games. Apart from this, the number of scene primitives varies significantly in the used scenes and ranges from 70k triangles up to 1 million triangles. The test results were produced on a GeForce GTX 1080, however, the relative performance is the same on similar systems. All data structures are implemented in the same environment and supported by acceleration in agreement. Hence, a fair comparison of the used structures is guaranteed. The resolution of the renderings was in all cases 720p. Primary and shadow rays were rendered with fast rasterization accelerated by a binary LS techniques as proposed in chapter 5 and by [Billen and Dutré, 2016] and are therefore out of the scope of this chapter. Regarding this, the shown approach accelerates calculations of all indirect lighting effects, resulting for example in ambient occlusion, diffuse illumination and glossy reflections.

Table 6.1 shows the quantitative results using the two main data structures with and without the acceleration of the LS with the mentioned integration depth parameter d. The build time, the memory consumption and the performance in ray tracing for indirect rays are evaluated. Obviously, the computation time and memory consumption of the LS need to be summed up on the values of the base data structure. The illustrated LS values in the table are already combined and therefore show the total sum. Moreover, the build times and the memory consumption of the LS significantly scale with the total number of computed LSs and not the number of scene triangles. For BVH initialization a binned SAH construction is used, resulting in good quality but non-interactive build times. The used build algorithm significantly affects the build time of the BVH, but as this work focuses on the relative comparison of the data structure with the usage of the LS, this does not affect the results.

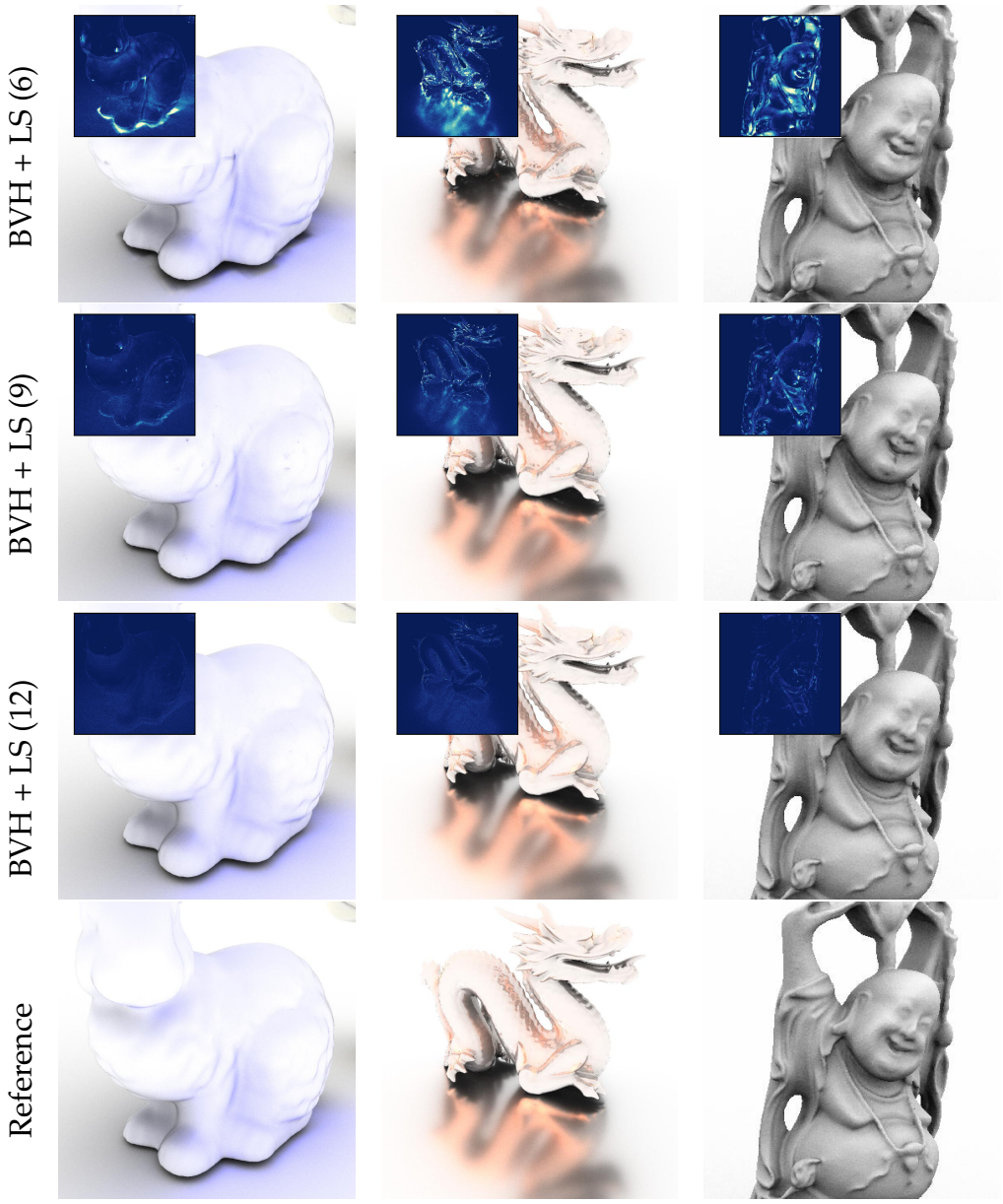


Figure 6.6: Some of the indirect illumination results of the evaluation. As illustrated, the buddha scene especially focuses on ambient occlusion, the bunny scene on diffuse materials and the dragon scene on glossy reflections. Nevertheless, all visual effects were indifferently produced with the same technique with the only difference in the object material. Therefore, the results are not optimized to show specific effects. The maximum LS integration depth d within the BVH is shown in $LS(d)$. RMSE error values per pixel are illustrated through the heatmaps where bright areas mark high error values.

Scene		BVH pure	BVH LS (9)	BVH LS (12)	NT (6,3) pure	NT (6,3) LS (2)	NT (10,3) pure	NT (10,3) LS (2)
BUNNY (69k tris)	**Init (s)**	0.3	2.2	10.4	3.7	9.6	17.2	44.3
	Size (MB)	5.5	38.7	227.2	1.5	17.6	23.9	149.7
	Error (RMSE)	–	0.028	0.011	–	0.032	–	0.022
	Perf (FPS)	76.7	194.7	131	20.6	89.4	36.7	63
	Perf (rel.)	**1.0x**	**2.5x**	**1.7x**	**1.0x**	**4.3x**	**1.0x**	**1.7x**
DRAGON (871k tris)	**Init (s)**	1.7	5.6	17.1	22.5	50.3	100.6	209.7
	Size (MB)	35.5	74.8	280	4.1	10.8	13.2	65.5
	Error (RMSE)	–	0.038	0.017	–	0.036	–	0.025
	Perf (FPS)	45.9	169.4	110.2	1.4	107.9	11.7	62.1
	Perf (rel.)	**1.0x**	**3.7x**	**2.4x**	**1.0x**	**77.1x**	**1.0x**	**5.3x**
BUDDHA (1087k tris)	**Init (s)**	2	6.2	17.6	27.6	61.6	123.2	254.2
	Size (MB)	40.7	80	285	4.5	10.9	11.6	57.5
	Error (RMSE)	–	0.021	0.007	–	0.030	–	0.019
	Perf (FPS)	62.4	195.4	128	1.1	121.4	12.1	69.4
	Perf (rel.)	**1.0x**	**3.1x**	**2.1x**	**1.0x**	**110.4x**	**1.0x**	**5.7x**
SIBENIK (75k tris)	**Init (s)**	0.1	1.9	7.4	2.3	18.6	13	102.3
	Size (MB)	3	57.7	251.6	8.3	286.4	106.9	2114.3
	Error (RMSE)	–	0.054	0.044	–	0.052	–	0.052
	Perf (FPS)	19.3	38.3	28.4	8.8	23.1	8.6	12.8
	Perf (rel.)	**1.0x**	**2x**	**1.5x**	**1.0x**	**2.6x**	**1.0x**	**1.5x**
SPONZA (262k tris)	**Init (s)**	0.5	3	9.2	7.1	28.5	36.3	153
	Size (MB)	10.1	54.1	231.2	8.9	347.6	133.3	2680.4
	Error (RMSE)	–	0.167	0.077	–	0.074	–	0.060
	Perf (FPS)	16.7	48.5	33.2	5.8	27.4	10.1	16.2
	Perf (rel.)	**1.0x**	**2.9x**	**2x**	**1.0x**	**4.7x**	**1.0x**	**1.6x**
CONFERENCE (331k tris)	**Init (s)**	0.7	3.2	9.6	7.3	27	36.8	115.2
	Size (MB)	13.4	61.3	213.2	6.3	170.9	86.3	1014.9
	Error (RMSE)	–	0.088	0.055	–	0.063	–	0.051
	Perf (FPS)	16	44.6	33.6	1.4	26.3	8.4	20.7
	Perf (rel.)	**1.0x**	**2.8x**	**2.1x**	**1.0x**	**18.8x**	**1.0x**	**2.5x**

Table 6.1: Test results of the evaluation in terms of initialization time (in seconds), memory consumption (in MB) and ray tracing performance (in FPS and relative to base structure) for BVH and N-tree as base data structures without and with the usage of the LS with varying integration depth parameter d. LS values are already combined with the base structure. BVH initialization was optimized in terms of ray tracing performance and not initialization speed. The relative differences in comparison to the base data structure without LS acceleration are marked. The parameter set for the N-tree is given in NT(N, d) and for the LS in LS(d). The LS subdivision parameter N is set to 6 for the BVH and for N-tree integration to the according value.

The runtime performance was measured in frames per second only counting indirect illumination. Apart from absolute values, the relative differences between pure and LS supported data structures show the benefit of the presented approach (illustrated in Figure 6.7). The BVH performance is near state-of-the-art in ray tracing performance. It is mainly regulated by the quality of the underlying tree and can be further optimized by recent techniques as proposed by [Ganestam et al., 2015] and [Yin and Li, 2014]. Using the technique presented here gives a significantly better performance depending on the used parameter set, however, with approximated results. This is due to the simplification of shaft data, where only a single candidate is stored and used for all rays passing a given shaft. Moreover, it is noticeable that the usage of the LS accelerates the data structure to an acceptable level, especially in those cases where the base data structure performs very poorly, as demonstrated in cases with a bad N-tree resolution. Overall, the incorporation of the LS accelerated a state-of-the-art BVH with a factor of more than $3\times$ and the N-tree of up to more than an order of magnitude.

Figure 6.6 and Figure 6.8 show the qualitative differences of various integration depths used in LS accelerated BVH in comparison to ground truth data. It is observable that lower parameter sets result in visible artifacts due to shaft simplification. However, by using higher integration depths for the LS within the BVH hierarchy the artifacts become manageable. The artifacts are especially noticeable in bigger scenes when the camera is positioned close to an object. This is mostly observable in the architectural scenes as shown in Figure 6.10. There, also the N-tree results with the LS are shown. As with the BVH, the quality of the LS accelerated N-tree improves when the LS is used in a deeper level of the tree hierarchy. Because of its regular structure, the N-tree LS is mostly better suited for big architectural scenes and produces less approximation artifacts for these cases in comparison to the BVH LS. However, as it was shown by [Yu et al., 2009], correctness in indirect illumination is not required and approximations are sufficient in most cases. Following this the BVH LS has better performance with mostly sufficient quality.

The main parameter for quantitative and qualitative analysis is the value of the used integration depth parameter d where LSs are created and used. A deeper integration depth results in more LSs, therefore causing higher build time and memory consumption, as well as lower ray tracing performance. This is due to the fact, that with a deeper LS integration depth more nodes in the hierarchy need to be traversed. However, the quality gets better when more LSs are used. With this the LS integration depth can be used as an arbitrary parameter for setting a trade-off between quality and performance. This is especially true, when the base data structure produces

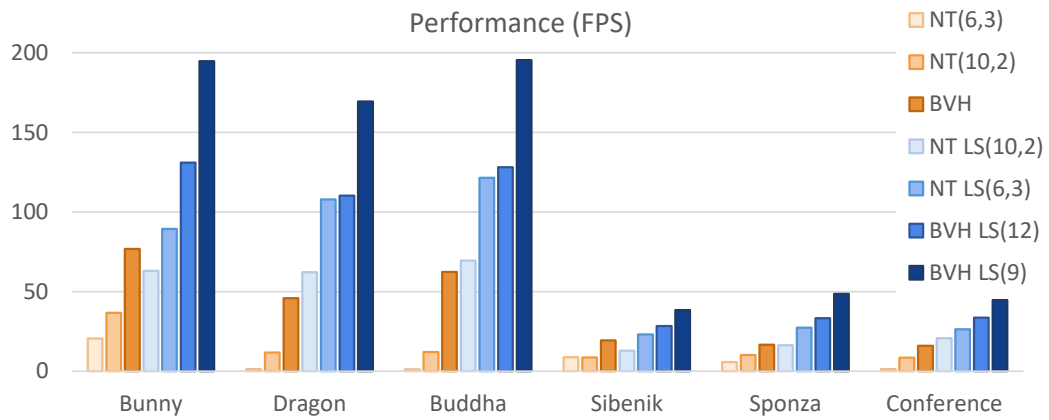


Figure 6.7: Performance illustration (in FPS, from Table 6.1) of the data structures without and with LS using two different integration depths each. It is visible that the LS is able to enhance performance significantly in all cases. Even when used in conjunction with a low-performance uniform grid, its performance is higher compared to a state-of-the-art BVH.

a deep tree hierarchy, as it is done with BVHs. The *N*-tree naturally only has a shallow tree hierarchy, therefore is not that suitable for a dynamic trade-off. Also, the integration depth within a binary tree like the BVH already limits the total number of nodes that contain a LS. With this, the total memory size is nearly independent from the scene geometry and only depends on the integration depth. As such, the LS scales significantly better with higher number of scene primitives compared to typical spatial data structures. This is illustrated in Figure 6.9.

6.5 Conclusion and Future Work

The non-binary LS with visibility precomputation of scene information is presented, which stores a single candidate as a representative per shaft. With this work, the general approach of precomputing directional information per LS shaft in application of indirect and global illumination is explored. Through the representative candidate precomputation, the need for intersection tests during traversal could be eliminated as far as possible. Although this technique results in approximation artifacts if the integration depth parameter is not high enough, it was able to show that these errors are nearly non-perceivable in the context of indirect illumination. When compared to the base data structure, this technique results in higher mem-

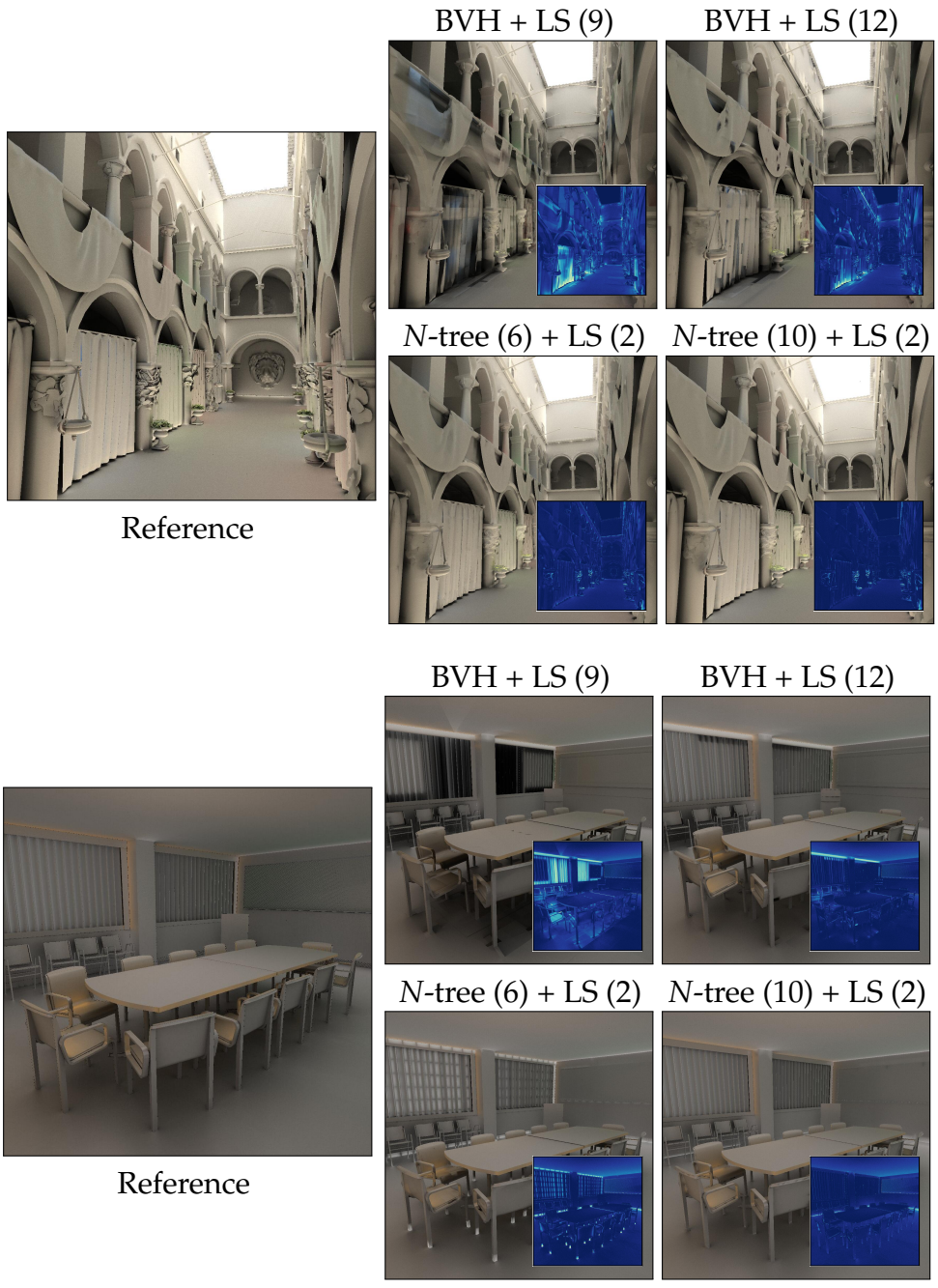


Figure 6.8: Test results with the architectural scenes. All images were rendered in 720p and only present indirect illumination. In bigger scenes using a lower parameter for the integration depth d of the LS usage within the base data structure, more approximation artifacts due to shaft simplification occur (brighter areas in the bottom right heatmaps).

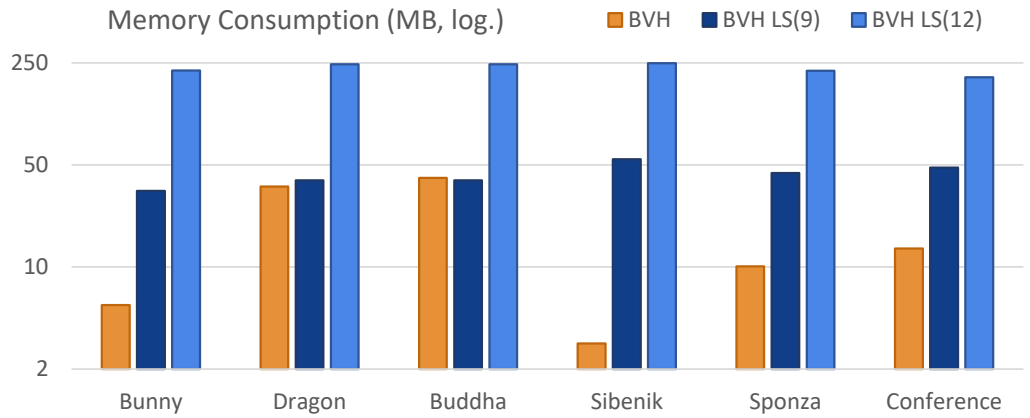


Figure 6.9: Memory consumption (in MB, from Table 6.1) of the BVH and the LS with two different integration depths. For the LS only directional information is presented, for the total needed memory the BVH size needs to be added. The logarithmic scaling shows that the needed memory of the LS is independent from the scene size and depends only on the used integration level. In contrast, the BVH memory consumption depends heavily on the number of scene primitives and therefore fluctuates immensely.

ory size and build time but is in all cases able to significantly surpass the base structure in terms of performance.

Moreover, a generalization of the LS to all spatial data structures based on bounding boxes was shown. It was demonstrated that with an adaptation to a state-of-the-art BVH, this resulted in higher performance in comparison to the N-tree, the previously used base data structure.

Future Work

The precomputation of scene information is only a beginning. By calculating and storing the lighting state in a shaft, it may be possible to use a great variety of rendering techniques without further computation overhead during traversal. Although this leads to a significantly increase in memory consumption, the gain in tracing performance can be a big improvement in path tracing scenarios. Using a dynamic combination with the base data structure has the potential to accelerate most ray tracing systems, using the base structure in situations where correct information is needed and the LS where fast approximated data is sufficient. An approach to further tackle the difficulties of memory size and initialization speed is to use the LS based on objects rather than the whole scene. With this each geometrical object has its own single LS. This grants the possibility for object instancing and a significant reduction of memory needed. Fur-

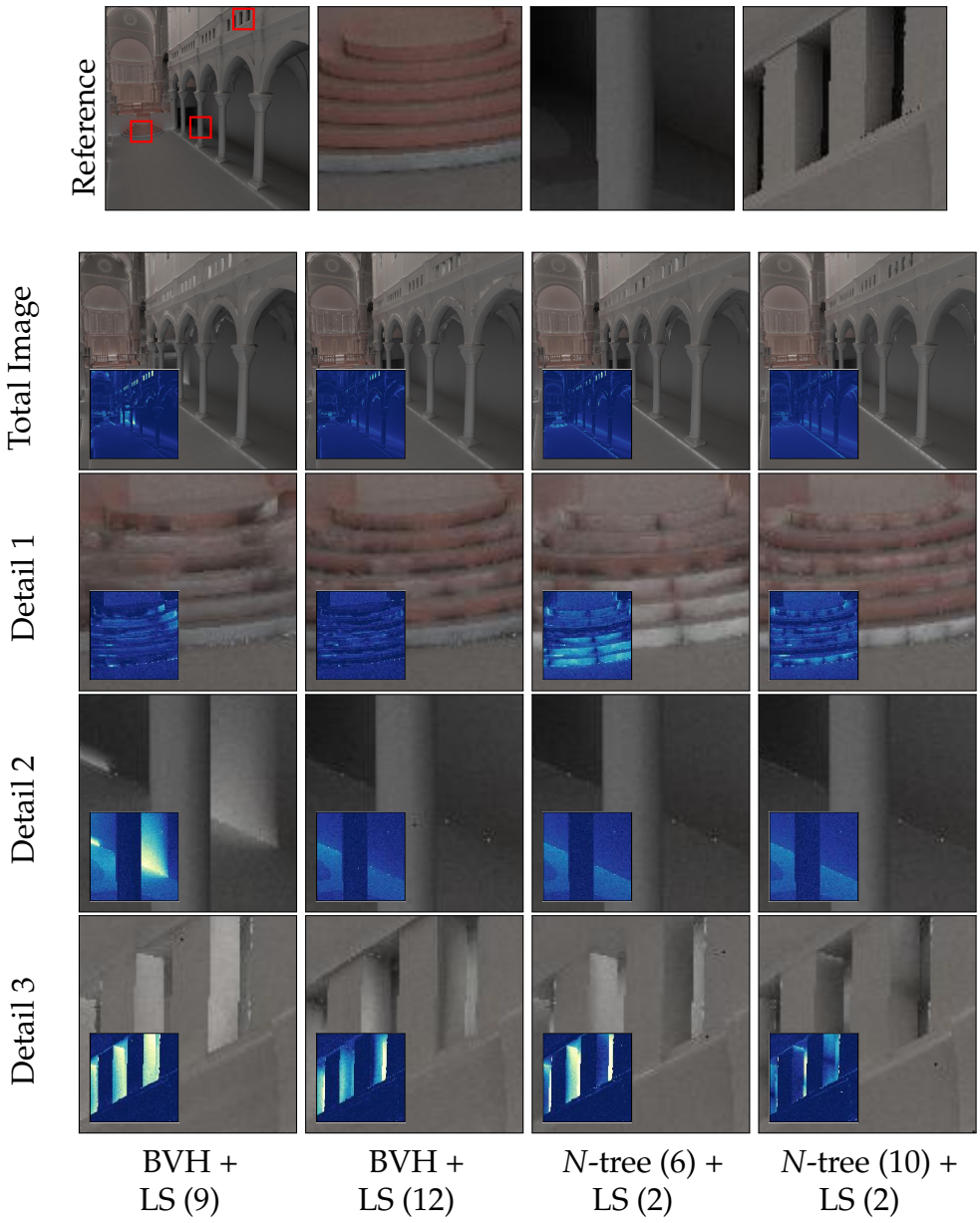


Figure 6.10: Detailed test results with an architectural scene showing indirect illumination. The magnifications and the heatmaps specifically show the weaknesses of the technique using a low integration depth parameter d. These artifacts especially occur in the transitions of different LSs. However, a deeper LS integration depth improves image quality significantly (darker areas in the heatmaps), making LS accelerated results suitable for indirect illumination. Overall, the perception in indirect illumination given a suitable integration depth parameter is mostly similar to ground truth renderings, but granting significantly better performance.

thermore, LS information is more accurate, solving the teapot in a stadium problem. Object LSs can be created beforehand, which significantly reduces initialization time. Combining the instancing aspect with local transformations creates the possibility to render dynamic scenes. Both, the dynamic combination with the base data structure in path tracing systems as well as object-specific LS usage, are further handled in chapter 7.

Another aspect that needs further investigation is the selection of the used parameter set. Currently, a fixed depth parameter is used for the whole LS tree, resulting in unnecessary high subdivision rate in sparsely filled areas. A dynamic subdivision scheme based on the number of candidates and the size of the current node would benefit the traversal and the LS accuracy.

CHAPTER 7

ADAPTION OF TWO-LEVEL STRUCTURES FOR PATH TRACING SYSTEMS

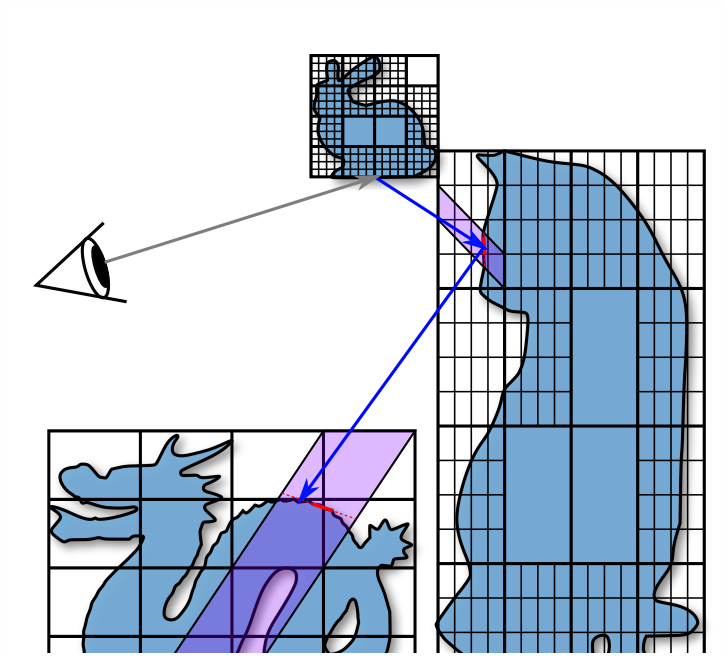


Figure 7.1: The LS adaption to two-level data structures improves accuracy of shaft predictions and further enables adaptive usage within path tracing systems. While early rays of a path are traced with data structures that grant slow but correct results, later rays are traced with the LS structure with different integration depths, granting fast performance but approximated results.

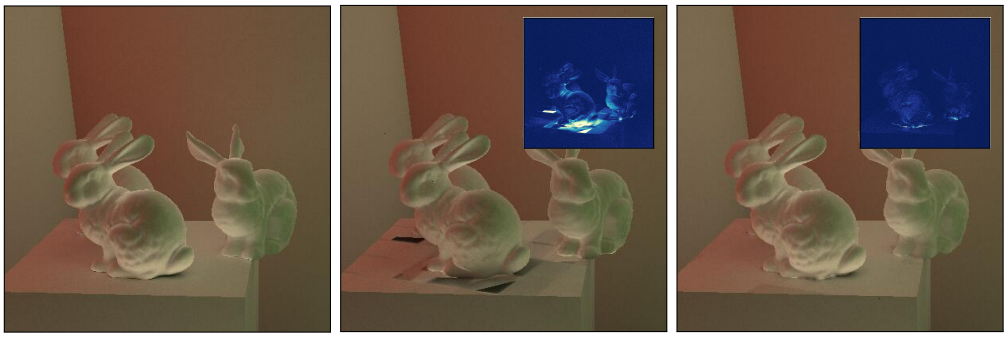
The contents of this chapter are based on the publication:

> Keul, K., Koß, T., Schröder, F. L., and Müller, S. (2019). Combining two-level data structures and line space precomputations to accelerate indirect illumination. In *Computer Vision, Imaging and Computer Graphics Theory and Applications*

7.1 Introduction

As mentioned in the beginning, the big vision is a structure with precomputed visibility information for all possible rays in order to reduce the ray traversal to a single lookup in some kind of table. However, as explained before, current hardware is not able to produce or utilize such a structure due to its immense memory consumption and initialization time. In this context, the LS was presented as directional data structure with precomputed visibility information based on the abstraction of shafts, with the main goal of reducing the traversal cost mostly through approximations during runtime. With this, the LS is one step further in the direction of the big vision. It was already shown to improve traversal of rays through empty space skipping in chapter 4. Significant improvements in performance were produced in terms of soft shadow computation in chapter 5 and indirect lighting and global illumination based on representative candidate precomputation per shaft in chapter 6. While the resulting images have visible artifacts due to the shaft approximations, it was shown that some artifacts are negligible when approximations are only used for indirect lighting computations [Yu et al., 2009]. However, limitations in terms of memory consumption and accuracy due to the shaft abstraction are visible. Furthermore, due to the precomputation, it is only possible to accelerate static non-interactive scenes until now. This chapter integrates the LS structure into two-level structures and thereby presents a way to reduce these downsides.

In the context of object instancing and rigid object movements, two-level data structures are traditionally used, which build a top-level structure around second-level object structures. For rigid object animations, only the top-level structure needs to be updated. However, the performance of two-level structures is in general inferior compared to their single-level flat counterparts, as was for example recently shown with BVHs [Benthin et al., 2017]. With this work, a combination of these two-level BVHs and representative candidate LS precomputations is proposed first. Typically, second-level object bounding volumes of two-level structures are small and tightly-fit to the contained objects. This property has the positive



Reference (BVH) F-LS (10) 2L-LS (4)

Figure 7.2: Improvements in indirect lighting calculation. A combined approach between two-level BVHs and approximation-based representative candidate LS (2L-LS) is presented. Even with a lower integration depth parameter (in brackets), less approximation errors compared to a LS integrated in a single-level flat BVH (F-LS) are generated (darker areas in the heatmaps). Additionally, less errors appear when used in later bounces.

effect of making LS approximations more accurate and reducing image errors, as illustrated in Figure 7.2. In addition, less LS nodes need to be generated and therefore less memory is consumed. Apart from that, the usability of the LS is significantly improved in scenes with object instancing, as the memory consuming parts of the LS can be reused for all instances of an object.

In a second step, a method for combining different data structures in path tracing systems is proposed. While pure BVH results are used on earlier path segments, where accurate results are needed, approximation-based LS results are used in later path segments, where accuracy is less relevant. By this, errors in the final image are nearly eliminated and the fast performance of the LS can be used almost without any drawbacks in image quality.

As indirect lighting calculation and ray tracing systems are extensively studied topics, they are well presented in previous work [Ritschel et al., 2012] [Pharr et al., 2016]. The field of acceleration structures for ray tracing has as well been researched intensively and a detailed overview of the most relevant literature in the most related topics of this chapter is presented in chapter 2. While there is a wide variety of acceleration data structures, the focus here is on bounding volume hierarchies (BVHs), as current research suggests, that these result in the highest ray tracing performance. For other spatial subdivision structures and comparisons,

it is referred to previous work [Havran, 2000] [Zlatuška and Havran, 2010] [Vinkler et al., 2016]. The main contributions of this chapter are:

- A fast but approximate technique for improvement of the representative candidate LS from chapter 6 through an efficient two-level hierarchy which additionally enables rigid transformations and object instancing.
- An efficient path tracing method that chooses different levels of LS abstraction to accelerate approximated intersection determination while also reducing the visible errors in the final image and thereby enhancing the structure's usefulness.
- An evaluation and parameter discussion in terms of memory consumption, achieved runtime performance and resulting image errors of the two-level technique.

7.2 Two-Level Line Space

The LS, as proposed in chapter 4 and chapter 5, is a considerable extension of typically used spatial structures. Its runtime performance is up to one magnitude faster than state-of-the-art BVHs, however, with the cost of higher memory consumption and initialization time. Moreover, it introduces tracing approximations leading to visible errors in resulting images. Therefore, it is favorable to reduce these approximation errors as well as the needed memory. An observation, used in this work, is that approximation accuracy is higher and less LS nodes are needed, when smaller and more tightly fitting bounding volumes are used for the LS. For this purpose, the LS is adapted to two-level BVHs that grant smaller bounding volume sizes in second-level object BVHs, leading to higher LS precision.

In that sense, first the representative candidate LS technique as proposed in chapter 6 is shortly revised, as it is used in this chapter. Afterwards, the adaption to two-level BVHs is described, which is one of the main contributions here. Lastly, a short discussion of all parameters is involved.

7.2.1 Limitations of the Representative Candidate Line Space

Tracing cost is dividable in two factors: the traversal cost of the underlying tree and the cost of the intersection tests that need to be made to find the intersected scene primitive. The representative candidate LS, as proposed

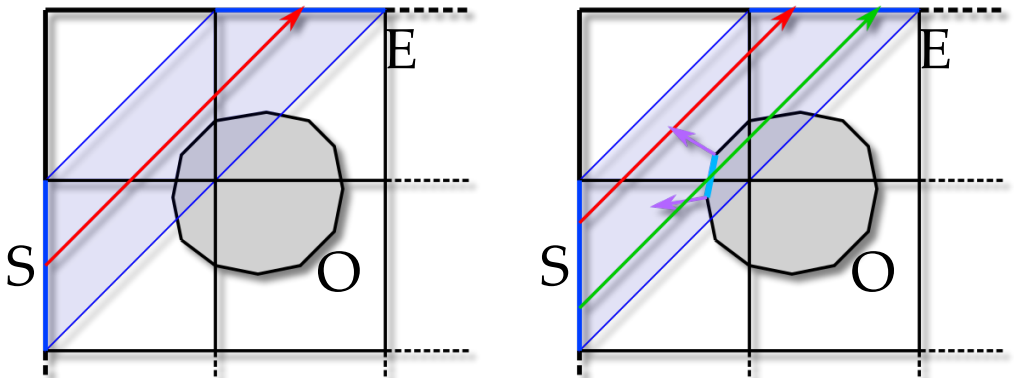


Figure 7.3: Initialization with representative shaft candidate search. If the midpoint ray of the shaft between patches S and E has no intersection (left), further rays are used (right) and the light blue candidate with violet normals is found.

in chapter 6, is based on the concept of finding representative intersection candidates for shafts during initialization. It reduces tracing cost in two ways. First, the tree traversal is cut at a specific level, the integration depth d where the representative candidate LS nodes are integrated, so that no further investigation of the tree is needed. With that, the tree traversal cost is limited to a fixed amount. Then, instead of calculating intersection tests between a ray and geometric scene primitives, the precomputed representative shaft candidate is used to approximate the ray intersection within that shaft, which results in a significant reduction of the intersection cost. While this leads to a remarkable gain in performance, the quality of the candidate approximations is, depending on the used parameter set, quite poor. This is especially visible in large architectural scenes, where a lot of scene primitives are grouped in mostly narrow spaces. There, a huge number of shafts need to be generated to cover the complete scene, which results in an extensive memory consumption. An insufficient parameter set results in big LS shafts and a significant approximation error, which potentially renders the data structure useless. Using a deeper LS integration depth within the BVH improves accuracy slightly, but at the cost of lower performance and much higher memory consumption. Keeping this in mind, the main goal is to improve approximation quality while also reducing the number of LS nodes needed.

The initialization of the representative candidate LS is straightforward. First, the underlying spatial data structure (i.e. the BVH) is constructed. Then a LS node is constructed for every node in a given depth of the underlying BVH. A LS node, as stated in chapter 4, is a bounding volume, where all sides of the bounding box surface are subdivided into equally

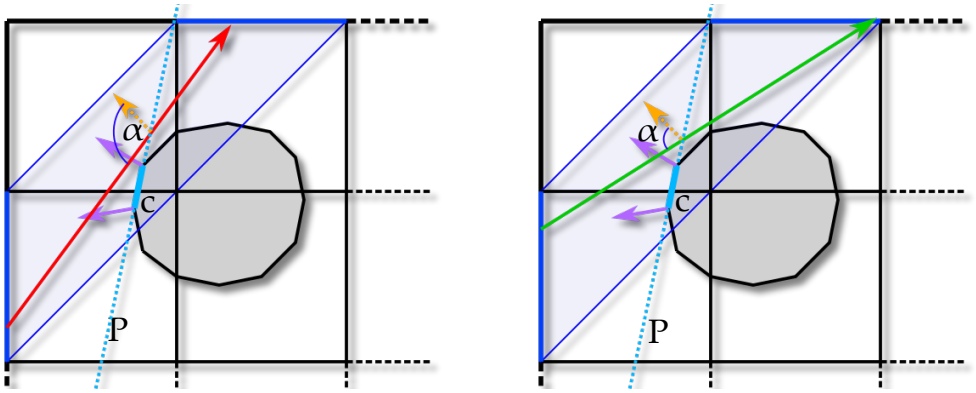


Figure 7.4: Runtime usage of the shaft candidate c. It is used as "infinite plane" P for all rays of the shaft. If angle α between a ray and the extrapolated normal (yellow) is too big, blocker approximation is refused (left). Otherwise it is used, even with no actual object intersection (right).

sized patches. Every two distinctive patches are joined to result in a shaft connecting those patches, which can also be seen as the accumulation of all possible rays that lead from one patch to the other. Hence, each ray through the bounding volume can be assigned to a single shaft. A LS node consists of all shafts within the bounding volume. In the representative candidate LS every shaft has a triangle reference, which describes the surface of the scene primitives covered by the shaft. The triangle (the representative candidate) is found by simple ray traversal within the underlying data structure during initialization. The ray used for traversal is determined by the midpoints of the patches that represent the shaft. If this ray has no intersection, additional rays can be traced. In this work, a regular arrangement of additional rays is used, i.e. the eight neighboring rays of the midpoint ray, but other schemes work as well. If no intersection was found, the shaft is declared as empty. This process is shown in Figure 7.3. During runtime, the representative candidate is used as a blocker approximation for all rays of the given shaft, resulting in an "infinite plane". However, if the extrapolated normal of the approximation exceeds a given threshold, a potential boundary of the contained object can be assumed, as illustrated in Figure 7.4. The approximation is then discarded and traversal continues.

7.2.2 Two-Level Line Space Structure

The main observation of this chapter is that LS approximations are considerably more accurate when used in smaller BVH nodes that tightly fit the contained scene objects. To benefit from this, the LS is combined with a

previously generated two-level BVH, that arranges logical scene objects on the first level and involves scene primitives only within the second level. Compared to flat BVHs, this scheme in general results in more complex BVHs with a higher total hierarchy depth and therefore reduced tracing performance. However, the second level object BVH nodes have a tighter fit to their content, which makes them smaller in size and more beneficial for the LS. The smaller size of object BVH nodes reduces the total number of LS nodes needed while simultaneously improving approximation accuracy. The results demonstrate, that the performance benefit gained through LS approximations outperforms traditional BVHs, even when used in two-level data structures. In that sense, four different setups of the data structure are differentiated, as also visualized in Figure 7.5:

1. Flat BVH, as state-of-the-art ray tracing structure;
2. Two-level BVH, better suited for rigid object dynamics and instancing;
3. Flat BVH with LS integration, for fast blocker approximations, as in chapter 6;
4. Two-level BVH with LS integration, same advantages as two-level BVHs and flat BVH-LS, with less approximation errors, presented in this work.

As before, the flat-BVH is the state-of-the-art competitor. Depending on the construction algorithm, it achieves the best runtime performance and initialization time. In comparison to these, two-level BVHs are optimized for rigid object dynamics and object instancing, at the cost of higher tree complexity and lower runtime performance. LS integration within flat BVHs was shown to achieve much better runtime performance due to the cut in the traversal hierarchy and the blocker approximation, since it is not necessary to calculate intersection tests. In contrast, the LS integration has a significantly higher memory consumption and longer initialization times. Apart from that, the blocker approximation leads to visible errors in the results.

The LS adaption into two-level BVHs combines the advantages of both techniques. There, second-level object BVHs contain LS nodes, which are integrated in a specific hierarchy depth. The top-level BVH is independent from this scheme and identical to the top-level hierarchy of traditional two-level BVHs. Similar to other two-level structures, the two-level LS is well suited for rigid object dynamics and instancing. This way, every unique logical object contains its own BVH-LS, whereas the top-level hierarchy stores

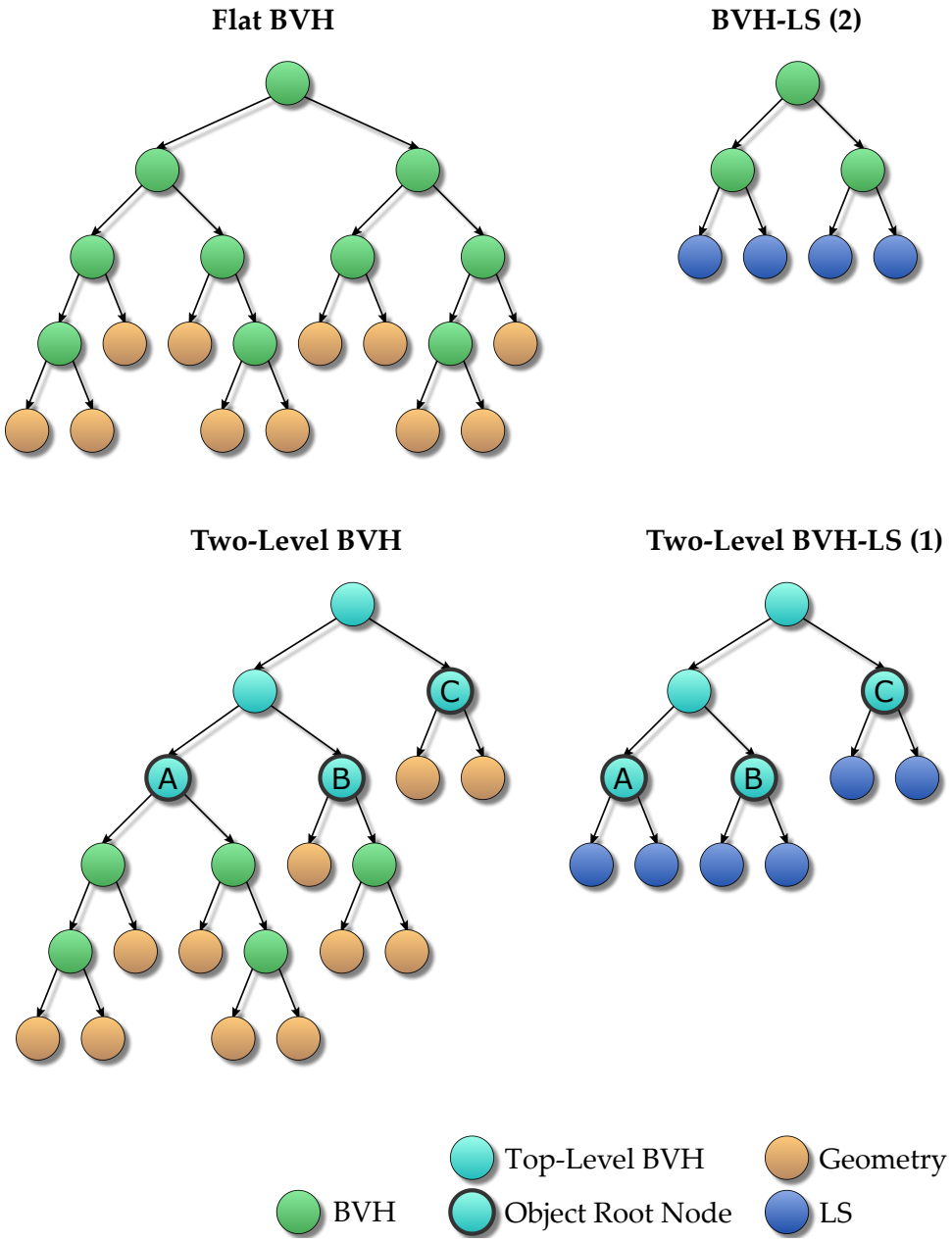


Figure 7.5: Acceleration structures used. Flat BVHs are the most common structures. Two-level BVHs are used in scenes with rigid object dynamics, granting minimal tree updates for object movements but worse runtime performance, mostly because of deeper tree structure. The LS and BVH combination was shown to improve performance due to early approximation-based traversal termination. This chapter combines LS and two-level structures, granting fast performance and higher approximation accuracy due to better object fitting. The LS numbers show the integration depth within (object-) hierarchy.

	Memory	Performance	Quality
F-BVH	★☆☆	★★☆☆	★★★
2L-BVH	★☆☆	★☆☆☆	★★★
F-LS	★★★	★★★★	★☆☆
2L-LS	★★☆	★★★☆	★★☆

Table 7.1: Expected results for the data structures. In terms of quality, flat BVHs and two-level BVHs give correct results, while LS techniques have better runtime performance.

references to those object structures in its leafs. Therefore, the two-level LS benefits significantly from object instancing, as every memory consuming LS is only stored once per geometrical object. Moreover, when objects are stored and reused in multiple scenes, the LS only needs to be constructed once, which accelerates initialization speed. Concerning hierarchy depth, the LS integration depth within object hierarchies of two-level BVHs does not need to be as high as in flat BVHs. However, the hierarchy of the top-level BVH needs to be considered additionally. With this, the total depth of two-level LS is in general higher than the depth of the corresponding flat LS, which results in a lower runtime performance. Nevertheless, as with the flat LS, a higher runtime performance compared to the corresponding flat and two-level BVHs can be expected. Because of the better fit to contained objects, the two-level LS has better approximation accuracy, even when less nodes are generated and less memory is consumed. These theoretical considerations are summarized in Table 7.1.

In terms of traversal performance, two additional aspects need to be considered. First, during ray traversal two-level structures potentially need to check multiple object nodes when overlapping objects are involved. A solution to this has been proposed recently, which opens and merges object BVHs dynamically during runtime [Benthin et al., 2017]. Second, as shown in Figure 7.5, two-level structures in general result in more unbalanced and partly deeper tree hierarchies compared to their flat counterpart. This is especially problematic when complex and simple objects are used simultaneously. During SIMT traversal, kernels are not able to benefit from thread coherency, which decelerates the overall traversal step. This technique represents a solution to this, as the LS limits the maximum object hierarchy depth to a specified level. This, along with the approximation-based early ray termination, results in better SIMT thread coherency due to less overhead caused by bad thread parallelism, which finally improves traversal performance.

Concerning BVH initialization, any of the existing construction or optimization algorithms can be used. Because of the long LS construction time, it grants no benefit to use a fast low-quality BVH builder for object BVHs. Moreover, a bad BVH quality in general results in larger bounding volumes, leading to bigger shafts and therefore worse LS approximation accuracy. Consequently, a high-quality BVH builder generates a lower number of nodes, that additionally have better scene coverage and are smaller in size. This generally grants better BVH performance and more accurate LS predictions due to smaller and more precise shafts. Hence, less LS nodes are needed for better quality and runtime performance. As the top-level BVH has no connection to the LS, it can be created with any build algorithm. Concerning object animations with two-level structures, it is plausible to use fast low-quality construction algorithms. In this setup, a state-of-the-art high-quality BVH builder as proposed in related work [Aila et al., 2012] is used for both, the top-level and object-level BVHs.

7.2.3 Parameter Discussion

Obviously, using an adequate parameter set is of significant importance for approximation quality, rendering performance and memory consumption. As before, there are two different parameters for this technique: the LS integration depth within the base structure and the subdivision parameter.

The integration depth d is the main parameter used in chapter 6. It represents the depth within the base data structure that stores LS nodes instead of children information. A higher depth results in a deeper hierarchy, more LS nodes and therefore higher memory consumption and lower runtime performance. However, image quality improves due to higher shaft precision, which leads to less approximation errors. This is illustrated in Figure 7.6. As mentioned, the LS integration depth in second-level BVHs does not need to be as high as in the flat LS to result in better quality, as shown in Figure 7.9.

The subdivision parameter N mainly determines LS accuracy. Shafts are created from the surface patches of the node's subdivided bounding volume. Higher subdivision leads to smaller patches and therefore thinner shafts, which provide more precise approximations. However, as already shown in chapter 4, significantly more shafts are generated and a more memory is consumed. As before, two different parameter values for N are used, that target either low memory consumption ($N = 6$) or higher approximation accuracy ($N = 10$). The runtime performance is mostly

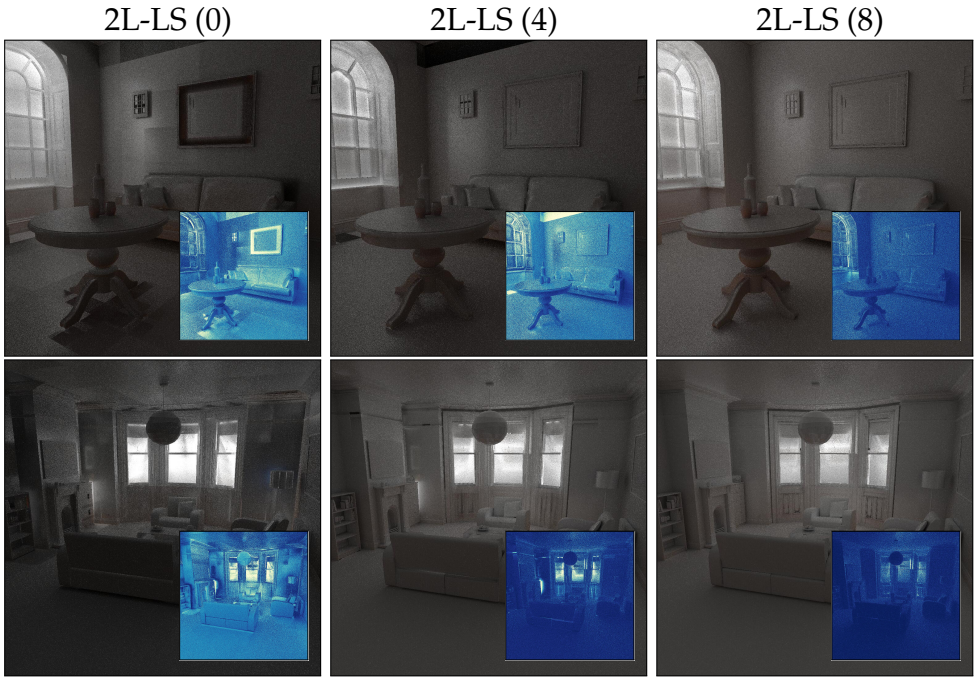


Figure 7.6: Indirect lighting results of different LS integration depths (marked in brackets - 0 means root node stores LS). Higher depths grant better approximations and less errors (darker areas in the bottom right heatmaps). Obviously, the needed integration depth is scene dependent.

unaffected by this parameter, as it has the same access time in all cases. The differences in quality are presented in Figure 7.9.

7.3 Path Tracing with multiple Data Structures

The early traversal termination with candidate approximation of the LS leads to a significant improvement in tracing performance, however, at the cost of approximation errors. By using an adequate parameter set, these errors can be greatly reduced, while in turn memory consumption increases tremendously. Keeping this in mind, the goal is to use different parameter sets and data structures in different segments of path tracing. With this, the BVH is used in situations that need correct results and the LS with different parameter sets depending on whether the focus lies on accuracy or tracing performance.

During path tracing, there are several ways to determine, whether a ray can be approximated or should be calculated accurately. It is for example

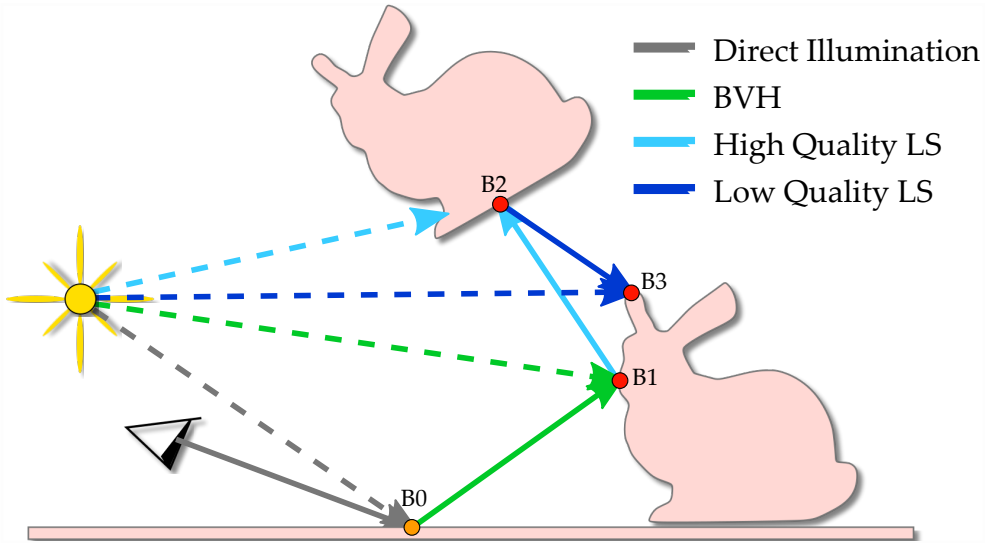
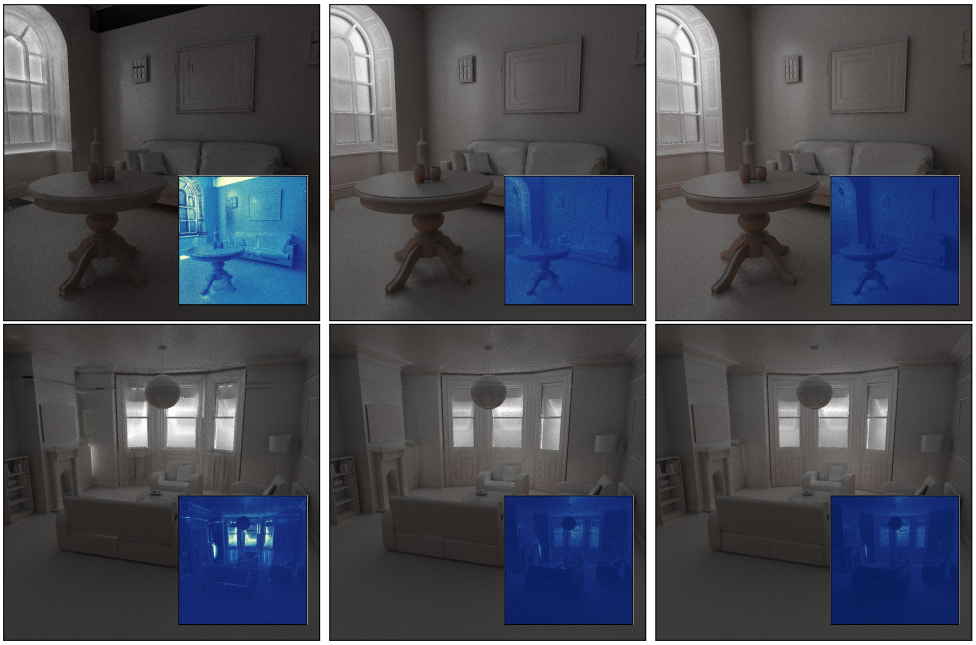


Figure 7.7: 3-bounce path using different data structures depending on the bounce depth. Primary and direct shadow rays use rasterization techniques. BVH path segments have precise results, while LS segments are traced faster.

possible to use the evaluation of the probability density function of the current path segment or to distinguish the type of the path segment, i.e. whether a shadow, reflected or diffuse ray is evaluated.

For the sake of simplicity, this distinction is done based on the bounce depth of the current path segment in a backward path tracing system. There, it is possible to correctly calculate the first bounce (indirect illumination through one object) and its according shadow ray with a BVH. The second bounce (indirect illumination with two intermediate objects) is less relevant and may be calculated with the faster approximation-based LS. Further rays have decreasing relevance in illumination and may therefore be calculated with a lower-quality higher-performance LS. This process is illustrated in Figure 7.7 and a demonstration with the resulting errors is shown in Figure 7.8. Primary rays and the according shadow rays can be rendered with rasterization-based approaches, which are fast and generate correct results.

In terms of deciding whether fast but approximated or accurate but slow results should be used, it is possible to use different other strategies. While the bounce depth is used here, an approach based on the distance to the bounding box is also reasonable. This approach is further explored in the usage heuristic, which is presented in chapter 8.



1. Bounce 2. Bounce 3. Bounce

Figure 7.8: Results of different bounce depths of LS usage. The path depth is shown, from which on approximation-based LS (2L-LS (4)) is used. Later usage of LS results in less errors (darker areas in the heatmaps) as well as less performance gain.

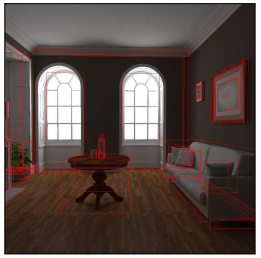
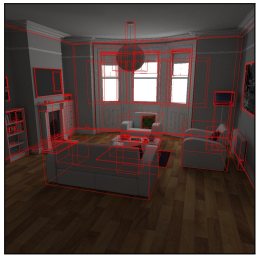
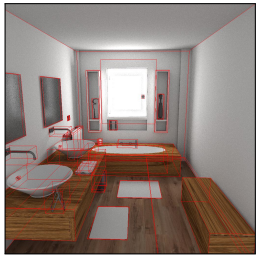
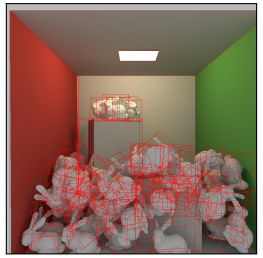
															
	FIREPLACE 143*k* triangles 49 objects			BREAKFAST 269*k* triangles 45 objects			LIVING ROOM 576*k* triangles 78 objects			SALLE DE BAIN 1231*k* triangles 39 objects			BUNNY SCENE 4467*k* triangles 64 instances		
Structure	**#LS**	**MB**	**s**	**#LS**	**MB**	**s**	**#LS**	**MB**	**s**	**#LS**	**MB**	**s**	**#LS**	**MB**	**s**
$N = 6, \approx 38k$ shafts per LS															
F-LS (8)	226	28.1	0.82	186	26.4	1.58	198	38.4	1.58	207	53.4	3.07	245	141.2	10.29
F-LS (10)	763	80.9	2.17	641	70.8	4.11	620	81.5	2.77	706	98.0	4.97	954	199.5	13.05
F-LS (12)	2223	206.0	5.83	2185	214.3	12.49	1936	199.1	6.33	2628	238.7	11.90	3750	407.4	23.78
2L-LS (0)	49	8.4	0.81	45	11.0	1.11	78	22.8	1.95	39	35.5	2.64	9	2.5	0.24
2L-LS (4)	526	53.4	1.62	648	68.5	2.89	940	109.2	3.60	467	76.1	3.87	24	3.8	0.29
2L-LS (8)	4195	344.9	8.27	7351	646.5	21.78	7373	631.7	15.92	6296	490.5	18.55	264	20.2	0.98
$N = 10, \approx 300k$ shafts per LS															
F-LS (4)	16	18.0	0.52	16	21.8	1.12	16	29.6	1.36	16	46.9	2.59	16	132.9	9.83
F-LS (6)	63	57.2	1.24	54	51.4	2.70	59	70.1	2.05	59	85.1	3.31	63	170.9	11.01
F-LS (8)	226	184.8	3.78	186	147.1	7.89	198	190.6	4.31	207	194.2	6.66	245	294.4	15.39
2L-LS (0)	49	38.0	1.24	45	35.8	2.05	78	74.7	2.71	39	62.3	3.21	9	7.0	0.30
2L-LS (4)	526	366.7	6.50	648	450.0	14.10	940	708.4	12.66	467	362.1	10.45	24	16.7	0.59
2L-LS (6)	1634	1058.4	19.62	2328	1549.6	47.93	2931	1926.1	37.25	1719	1079.0	32.82	72	42.6	1.49

Table 7.2: Construction results of the single-level flat BVH-LS (F-LS) and the two-level BVH-LS (2L-LS) in terms of the total number of generated LS, memory consumption (in MB) and build time (in s). The integration depths are shown in brackets.

7.4 Results

The two-level LS approach (2L-LS) as approximation-based method is tested and evaluated for indirect lighting calculation. The test setup is comparable to chapter 6, where the single-level flat BVH-LS (F-LS) was proposed, which is the main competitor in terms of performance. For a complete evaluation, it is also compared against the pure underlying structures (F-BVH and 2L-BVH), which produce approximation-free results. The evaluated data structures are shown in Figure 7.5. For all BVH algorithms state-of-the-art techniques targeting good tree quality and fast runtime performance are used, as shown in [Aila et al., 2012]. All methods are compared in terms of traversal performance, memory consumption and build time. Additionally, the approximation-based results are compared per bounce to ground truth data and show, that approximations produced by the presented technique are more precise than those of F-LS.

The chosen test scenes are commonly used architectural scenes (FIREPLACE ROOM, BREAKFAST ROOM, LIVING ROOM and SALLE DE BAIN) and a scene with instanced objects (BUNNY SCENE). These scenes consist of multiple manually separated objects, which are used as objects in the top-level BVH, as shown in Table 7.2. In general, these objects can be interactively animated with rigid transformations, as this only results in a simple recalculation of the top-level BVH without modification of the second-level object structures. A multiple-bounce path tracing system is used for indirect lighting calculation with different data structures that can be used in different path depths. In all cases an image resolution of 1024×1024 was used. Primary and direct shadow rays are not computed via ray tracing, as there are faster rasterization-based approaches. The test system consists of an Intel i7-6800k 3.6 GHz CPU and a NVidia GeForce GTX 1080 GPU using GLSL Compute Shaders. To guarantee a fair comparison of the used data structures, no third party implementations are used. Overall, the presented results confirm the theoretical considerations from subsection 7.2.2.

Table 7.2 shows the quantitative construction results of the presented method compared to the single-level LS. In both cases, two different subdivision parameters ($N = 6$ and $N = 10$) and three different integration depths are used, leading to mostly similar memory consumption for all data structures, which makes them comparable. Obviously, memory usage and build time are dependent on the total number of nodes containing LS information. Hence, scenes using object instancing have significantly less memory consumption and by far lower build times in two-level structures, as shown by the BUNNY SCENE.

Structure	Perf MRays/s	RMSE on Bounce 1	2	3	Perf MRays/s	RMSE on Bounce 1	2	3
	FIREPLACE				**BREAKFAST**			
F-BVH	98.3	–	–	–	40.8	–	–	–
2L-BVH	39.3	–	–	–	29.1	–	–	–
F-LS (8) , $N = 6$	340.6	.025	.010	.003	329.1	.048	.017	.004
F-LS (10) , $N = 6$	280.9	.024	.006	.002	290.9	.046	.017	.004
F-LS (12) , $N = 6$	243.2	.013	.003	.002	251.8	.045	.017	.004
2L-LS (0) , $N = 6$	249.9	.017	.003	.002	207.4	.024	.008	.002
2L-LS (4) , $N = 6$	123.9	.011	.002	.002	127.6	.020	.008	.002
2L-LS (8) , $N = 6$	93.1	.007	.002	.001	97.8	.012	.004	.001
F-LS (4) , $N = 10$	611.8	.023	.005	.002	462.4	.082	.029	.006
F-LS (6) , $N = 10$	403.7	.015	.004	.002	360.5	.071	.024	.005
F-LS (8) , $N = 10$	316.3	.008	.002	.002	303.5	.038	.014	.003
2L-LS (0) , $N = 10$	238.8	.010	.003	.002	202.2	.022	.006	.001
2L-LS (4) , $N = 10$	120.3	.008	.002	.001	128.1	.012	.003	.001
2L-LS (6) , $N = 10$	99.3	.004	.002	.001	114.1	.012	.003	.001
	LIVING ROOM				**SALLE DE BAIN**			
F-BVH	70.7	–	–	–	89.3	–	–	–
2L-BVH	27.7	–	–	–	45.8	–	–	–
F-LS (8) , $N = 6$	348.7	.028	.012	.005	479.1	.033	.011	.004
F-LS (10) , $N = 6$	279.3	.016	.004	.002	406.0	.030	.010	.003
F-LS (12) , $N = 6$	236.7	.013	.003	.002	361.0	.024	.007	.003
2L-LS (0) , $N = 6$	139.0	.038	.023	.010	262.2	.027	.006	.002
2L-LS (4) , $N = 6$	78.3	.009	.003	.002	156.6	.018	.005	.002
2L-LS (8) , $N = 6$	62.8	.004	.002	.002	124.5	.019	.005	.002
F-LS (4) , $N = 10$	651.5	.068	.040	.014	800.7	.023	.008	.004
F-LS (6) , $N = 10$	411.1	.039	.020	.007	546.8	.020	.005	.003
F-LS (8) , $N = 10$	307.6	.021	.008	.003	438.0	.014	.003	.002
2L-LS (0) , $N = 10$	136.6	.019	.007	.003	254.3	.019	.007	.004
2L-LS (4) , $N = 10$	75.0	.008	.003	.002	154.2	.012	.004	.001
2L-LS (6) , $N = 10$	65.3	.004	.002	.002	135.5	.011	.003	.001
	BUNNY SCENE				**AVERAGE OF ALL SCENES**			
F-BVH	54.6	–	–	–	70.7	–	–	–
2L-BVH	29.8	–	–	–	34.3	–	–	–
F-LS (8) , $N = 6$	387.5	.014	.006	.002	377.0	.030	.011	.004
F-LS (10) , $N = 6$	307.8	.008	.002	.001	313.0	.025	.008	.002
F-LS (12) , $N = 6$	248.6	.007	.002	.001	268.3	.021	.006	.002
2L-LS (0) , $N = 6$	181.4	.007	.002	.001	208.0	.023	.008	.003
2L-LS (4) , $N = 6$	114.6	.002	.001	.001	120.2	.012	.004	.002
2L-LS (8) , $N = 6$	84.5	.001	.001	.001	92.5	.009	.003	.002
F-LS (4) , $N = 10$	586.7	.023	.012	.004	622.6	.044	.019	.006
F-LS (6) , $N = 10$	480.7	.019	.009	.002	440.6	.033	.012	.004
F-LS (8) , $N = 10$	367.8	.010	.006	.002	346.6	.018	.007	.002
2L-LS (0) , $N = 10$	177.2	.004	.001	.001	201.8	.015	.005	.002
2L-LS (4) , $N = 10$	110.7	.001	.001	.001	117.7	.008	.003	.001
2L-LS (6) , $N = 10$	96.8	.001	.001	.001	102.2	.006	.002	.001

Table 7.3: Path tracing performance (in million rays per second - MRays/s) and per bounce error (in root-mean-square error - RMSE) of data structures with varying LS integration depth (in brackets) and subdivision parameter (value of N). In general, the 2L-LS results in a better trade-off between performance, image errors and memory consumption (see Table 7.2) compared to F-LS. It grants better performance in comparison to the base 2L-BVH and F-BVH and is a viable alternative to the F-BVH especially in scenes with rigid animations.

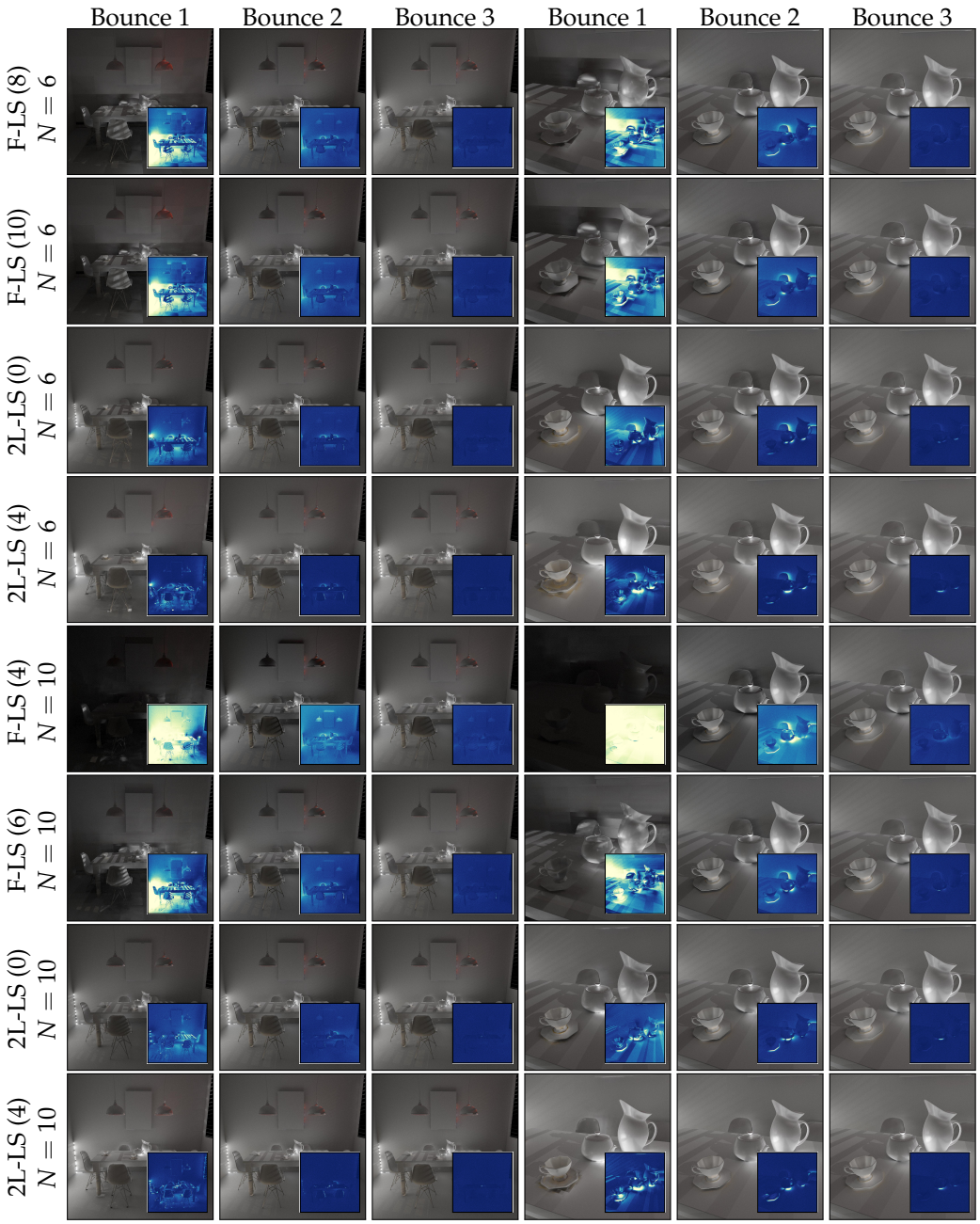


Figure 7.9: Per bounce results of F-LS and 2L-LS using different parameter sets. Obviously, higher integration depths (in brackets) and subdivision parameters (N) lead to less errors in the resulting images (darker areas in the bottom right heatmaps). In turn, higher parameters in general result in larger memory consumption, as illustrated by Table 7.2. The exact performance and error values are shown in Table 7.3.

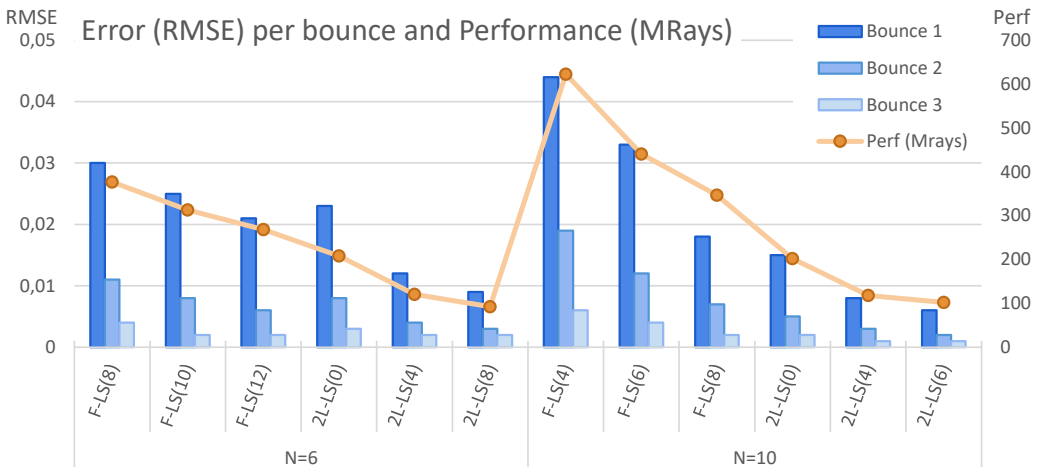


Figure 7.10: Average runtime performance (in orange, million rays per second, MRays) and per bounce error (in blue, root-mean-square error, RMSE) for the tested data structures from Table 7.3 . It is visible, that the achieved performance is proportional to the produced image error. The F-LS typically results in higher performance, while the 2L-LS produces less errors. As such, a combination of these approaches results in a dynamic trade-off between errors and performance.

Table 7.3 and Figure 7.2 as well as Figure 7.9 show runtime results of the data structures. There, tracing performance and error values for indirect rays are shown. The approximation-based errors produced by the LS are specified by RMSE (root-mean-squared error) values, which are based on the per-pixel differences to ground truth data. These are also presented in the table and the final images. The bounce depth marks the depth from which on the shown data structure is used - earlier bounces use the BVH for correct results. It is noticeable, that later bounces traced with approximation-based LS structures lead to insignificant errors but much faster tracing performance. It can be seen that two-level structures in general have inferior performance in comparison to their single-level flat counterparts, as explained in subsection 7.2.2. In turn, they produce less errors in general, even when a lower depth integration parameter is used. As visualized in Figure 7.10, the achieved runtime performance and the produced image error are proportional - higher performance results in equally higher image errors. This is true for both, the single-level LS as well as the two-level LS. The results also show that higher subdivision parameters need significantly more memory and therefore lower integration depths are used to limit the needed memory, which in turn reduces approximation accuracy leading to worse image results. Due to this, higher

Data Structure in bounce			Average of all scenes		
1	**2**	**3**	**MB**	**MRays/s**	**RMSE**
F-BVH	F-BVH	F-BVH	21.0	70.7	–
F-BVH	F-BVH	2L-LS 1	83.2	87.2	.002
F-BVH	2L-LS 1	2L-LS 1	83.2	103.7	.004
F-BVH	2L-LS 1	2L-LS 2	126.8	130.9	.004
F-BVH	2L-LS 2	2L-LS 2	64.6	158.1	.005
F-BVH	2L-LS 1	F-LS	140.7	189.3	.006
F-BVH	F-LS	F-LS	78.5	274.9	.011

Table 7.4: Summarized path tracing results (Memory in MB, Performance in MRays/s and Error in RMSE) with different data structures for different path segments averaged over all scenes (see Table 7.2 and Table 7.3). Used LS structures are: 2L-LS 1 = 2L-LS(4) $N = 6$, 2L-LS 2 = 2L-LS(0) $N = 10$, F-LS = F-LS(8) $N = 6$.

subdivision parameters are more suitable for later path segments, in which the lower integration depth leads to better performance, while the higher subdivision preserves accuracy to a higher degree. Summarized results for different data structures used in different bounces, averaged over all scenes, are shown in Table 7.4.

7.5 Conclusion

A novel combination of typically used two-level BVHs and LS approximations are proposed. It is explained how the combination benefits from both of their advantages while at the same time reducing their disadvantages. The object-level LS reduces the approximation error of the LS and increases the tracing performance by limiting the integration depth. Consequently, the presented approach results in better runtime performance compared to the traditional BVH with less memory consumption and approximation errors compared to the single-level LS structure. Furthermore, a way to connect the strengths of different data structures in a multiple-bounce path tracing system is presented, with a per bounce selection that indicates, whether the accurate BVH or faster approximation-based LS structure is used.

In an extensive evaluation the usefulness of these approaches is presented, showing different integration depths, subdivision parameters and per bounce results for the flat BVH-LS and two-level BVH-LS combination

and compared the results with state-of-the-art techniques. In that, the evaluation shows that the two-level approach is an improvement of the flat technique. Moreover, the two-level approach makes LS approximations usable in scenes with rigid object animations and especially useful coupled with object instancing. Finally, the per bounce usage reduces approximation errors at a high level, while at the same time enables better LS usage in combination with traditional data structures.

For future work, the idea of combining different LS approximation levels with correct BVH results can be further explored. For now the differentiation between those structures is based on the path depth of the path tracing system. The two-level approach allows for a refined differentiation based on a heuristic that involves the intersection information of object bounding boxes with the current ray. This is further handled in chapter 8 with the introduction of the usage heuristic.

CHAPTER 8

EFFICIENT PRECOMPUTED VISIBILITY FOR ACCELERATION OF OCCLUSION DETERMINATION IN TWO-LEVEL STRUCTURES

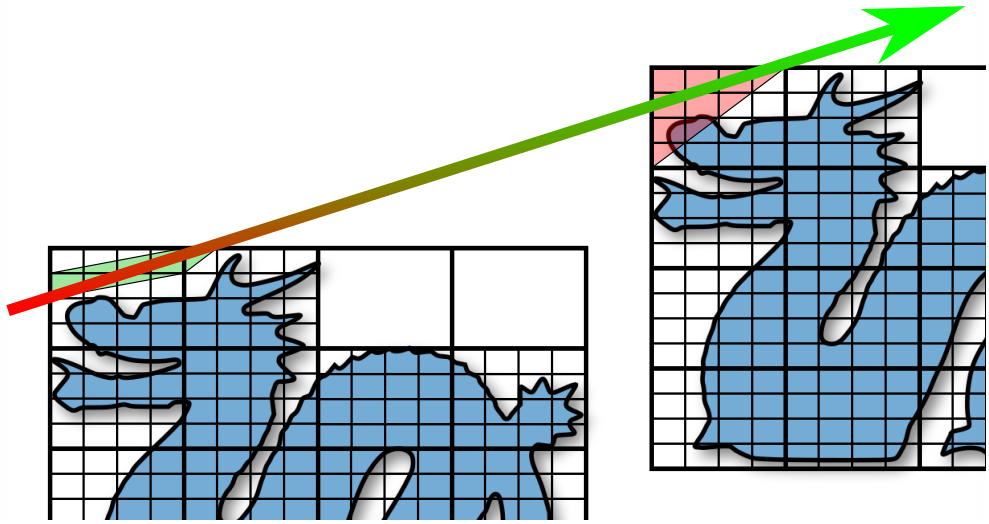


Figure 8.1: The usage heuristic enables the possibility to evaluate the significance of LS approximation errors. In near objects the results need to be correct (as illustrated by the red part of the ray), while for far away objects approximated results are sufficient (green part of the ray). This enables dynamic decision of whether correct but slow BVH traversal or fast but approximated LS termination is used.

8.1 Introduction

In the big vision of complete visibility precomputation for fast ray traversal computation with minimal cost during runtime, the LS was presented as viable structure. By integration into traditional spatial base data structures, it achieves good results with soft shadow and indirect illumination approximations through binary emptiness and representative candidate precomputation, as demonstrated in chapter 5 and chapter 6. To reduce the downsides in terms of memory consumption and accuracy as well as enable the support of interactive scenes with rigid object transformations, the integration into two-level structures was proposed in chapter 7. Additionally, a strategy was presented there, to decide whether accurate or fast approximated results are sufficient to use based on the current path depth in path tracing systems. However, this strategy is quite simple and the two-level approach gives more opportunities. In that context, this chapter presents a more elaborate strategy based on two-level structures and a simple usage heuristic, which decides whether to use accurate or fast results with varying LS integration levels. For simplicity, this is applied to shadow rays, similar to chapter 5, but the main idea is easily applicable for other types of rays.

The main observation is that the LS approximations are sufficient in quality in soft shadow computation when the intersecting object is small in size or far away from the current point. A big object near to the shadowed point leads to large LS shafts and therefore heavy approximation errors. Similarly, a small light source which is near to the intersecting object is more critical in terms of shadow approximation, while big area light sources that are far away from the intersecting object lead to insignificant approximation artifacts. These observations are used in order to propose a combined data structure with LS approximations that are used in conjunction to the accurate base data structure. A simple heuristic is introduced which decides whether it is necessary to use accurate results or if it is feasible to benefit from higher performance of LS approximations, as demonstrated in Figure 8.2. It is further observed that the LS already abstracts shadow samples of a specific area of the object, which makes it especially beneficial for ambient occlusion calculation, however with noticeable artifacts at the boundaries of the LS's bounding box. To reduce these artifacts and benefit from high LS performance in ambient occlusion calculation, a technique is proposed to generate shadow samples not from the point of interest but sample points based on a disc sampling technique, where the disc is aligned with the surface normal of the given point. This reduces image artifacts to a nearly non-visible amount and further improves the structure's usability.

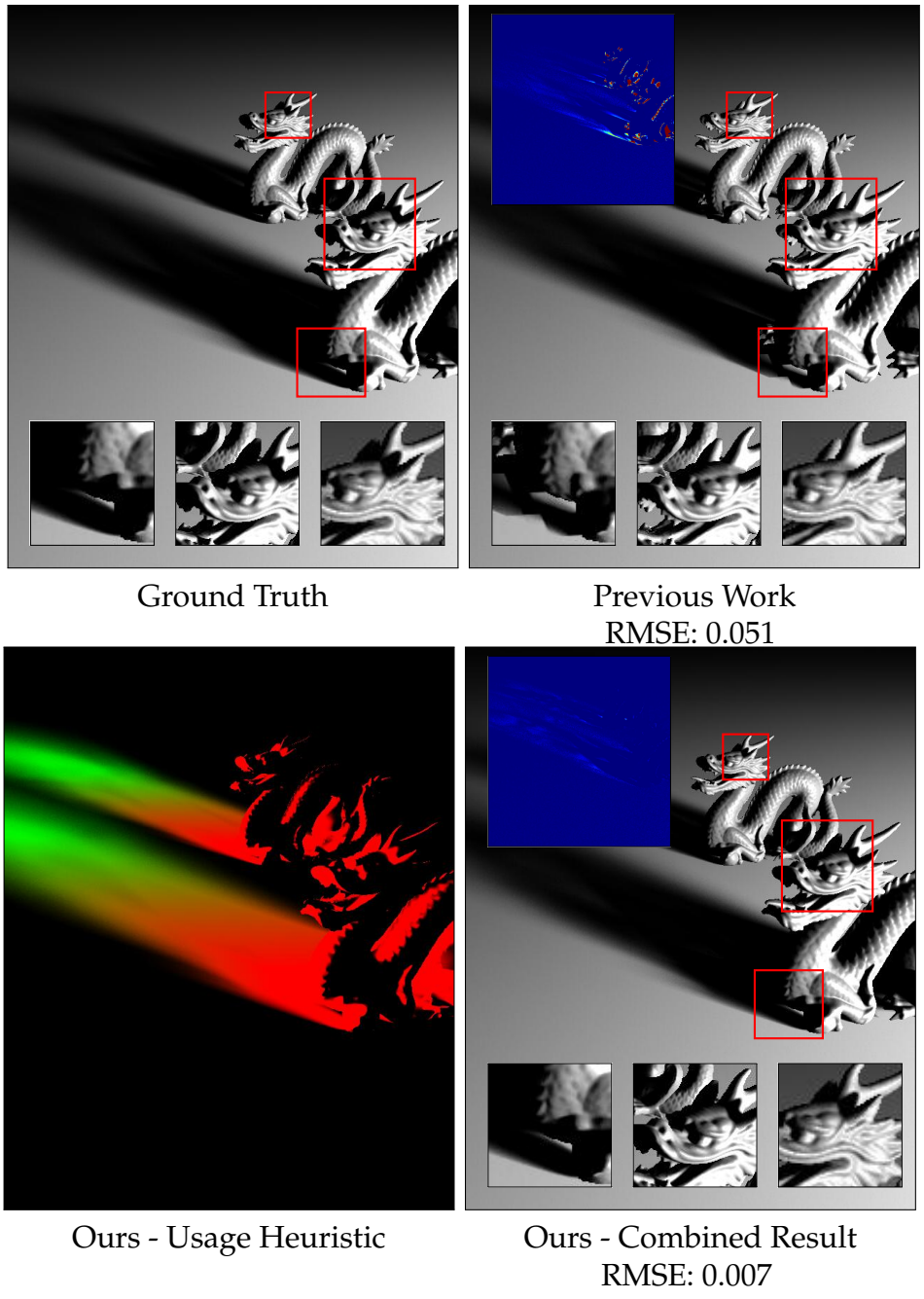


Figure 8.2: Overview of this work. While previous work on LS occlusion determination (chapter 5) suffers from high approximation errors, this is reduced by an adaptive design through incorporation of the usage heuristic. Red areas in the usage heuristic illustration are traced by the pure BVH in order to achieve correct results, while green areas indicate that LS approximations are sufficient. With this, the results dynamically combine fast LS approximations and accurate results to decrease errors while maintaining high performance.

In short, the contributions are:

- A simple usage heuristic that decides whether to use blocker approximations or accurate results
- An adaptive two-level LS approach for fast occlusion determination in soft shadow and ambient occlusion computations with reduced image artifacts
- A disc sampling strategy to reduce artifacts created in ambient occlusion calculation

8.2 Related Work

The generation of shadows and ambient occlusion are vastly studied topics with a great amount of literature. Therefore only a short overview of the most necessary and closely related work to this approach is presented. For further information, it is referred to the listed literature. A detailed description of the necessary acceleration structures for ray tracing is presented in chapter 2.

Shadow Generation

Shadow algorithms are typically divided in three different approaches: Image based reprojection with shadow mapping, geometry based shadow volumes and occlusion detection with ray tracing. The most frequently used technique in real-time graphics builds on image based projection [Williams, 1978]. For every light source a shadow map is generated, containing the foremost scene information seen from the light source. During runtime, every point visible from the camera is reprojected to the shadow map to indicate whether this point is nearest to the light source or blocked and therefore shadowed by an intermediate object. The quality of this method was improved by multiple shadow map lookups with percentage closer filter, which was later extended for soft shadow generation with area lights [Fernando, 2005]. A different approach is to store the squared depth additionally to the typical depth in order to use the Chebychev inequality to estimate the variance and thereby compute a soft shadow approximation, which is called variance shadow mapping (VSM) [Donnelly and Lauritzen, 2006]. This was extended to variance soft shadow mapping (VSSM), where average blocker distance is computed in order to achieve better results [Yang et al., 2010]. In fact the main idea sounds somewhat similar to the approach presented in this chapter but is

different in the main realization. In the previous work, the variance as well as the average blocker distance are used to calculate the soft shadow's area size. In the approach presented in this chapter, the blocker size and distance are used to calculate the usage heuristic and thereby the actual LS integration depth that is used for blocker approximation. This speeds up the ray traversal process, but nevertheless more shadow samples are needed compared to the VSM technique. Then again, as with other real-time techniques, only the samples stored in the shadow map and not the complete scene information is used with VSM and therefore errors are incorporated in the final image. This is not the case with ray tracing techniques, as used with the LS.

A different approach suitable for real-time rendering is done by generation of shadow volumes, that are created by extending the scene geometry from the source's center point into infinity [Crow, 1977]. Contrary to shadow mapping, this approach is accurate in shadow generation from a point light source, however more computing extensive due to worse scaling with the amount of scene primitives (e.g. triangles). Again, extensions for soft shadow generation were proposed [Laine et al., 2005]. However, in terms of image quality in soft shadow generation, all real-time approaches use some sort of approximation, which leads to inaccurate results. As this work focuses on acceleration and visibility precomputation of ray tracing, it is referred to [Hasenfratz et al., 2003] [Eisemann et al., 2011] and [Li and Liu, 2018] for further information on real-time shadow generation. Ray tracing describes the general process of image creation by searching the first intersection point in a given direction. This can be used for shadow generation by creating a ray starting from the regarded point, following the direction to the light source and determining whether a blocking object is in front of the light source. It is easily extended to area lights and accurate soft shadow computation by using multiple rays as shadow samples with different points of the light source's surface. More information on ray tracing is presented in [Pharr et al., 2016].

Ambient Occlusion

Apart from shadow computations based on light sources, ambient occlusion (AO) is an approximation of the global illumination for diffuse surfaces. For a detailed overview in the topic of AO it is referred to [Méndez-Feliu and Sbert, 2009]. One of the fastest and most popular approaches in real-time applications is based on approximations in image screen space, where the depth of multiple surrounding samples is compared [Shanmugam and Arikan, 2007]. Correct AO is often computed with

ray tracing and mostly done in a preprocessing step. With the use of voxel cone tracing [Crassin et al., 2011] it was shown that those effects can be approximated well in real time applications.

8.3 Adaptive Line Space for Blocker Approximation

This chapter builds on top of the two-level LS structures from chapter 7 and previously introduced blocker determination through LS approximation from chapter 5. Here, it is shown that the two-level approach is able to significantly reduce the shadow artifacts in soft shadow approximations. Furthermore, an adaptive technique is proposed for deciding whether the approximation based acceleration of the LS or the accurate BVH sould be used for tracing based on a simple heuristic. Additionally, an extension to ambient occlusion calculation is shown, which reduces the resulting artifact to nearly non-visible amount.

8.3.1 Two-level Line Space Precomputations and Soft Shadow Approximation

First, a short revision and reformulation of the main principles in LS visibility precomputation is presented. It is a ray classification scheme that builds on an intermediate level of shafts. The surface areas of every single bounding box are subdivided into $N * N$ equally sized patches. The shafts are produced with these patches, so that every pair of two patches represent the two opposite ends of a shaft. Every ray that intersects this bounding box can be assigned to a specific start and end patch and with that to a single shaft. Hence, every shaft serves as a proxy structure for all possible rays that are contained by it, and can be assigned with certain visibility attributes, which can be applied to the contained rays. The simplest attribute is the emptiness of the shaft that can be used for empty space skipping of ray traversal, due to an early ray termination. Inversely, a non-empty shaft can be interpreted as blocker determination approximation and shadow estimation, which is one of the core idea embedded in the work at hand. Previous work also implemented a representative candidate per shaft that was applied as blocker approximation and used in indirect lighting and path tracing calculation. While this is beneficial for runtime performance, it results in high memory consumption and noticeable image artifacts. The adaption to two-level structures resulted in higher approximation precision

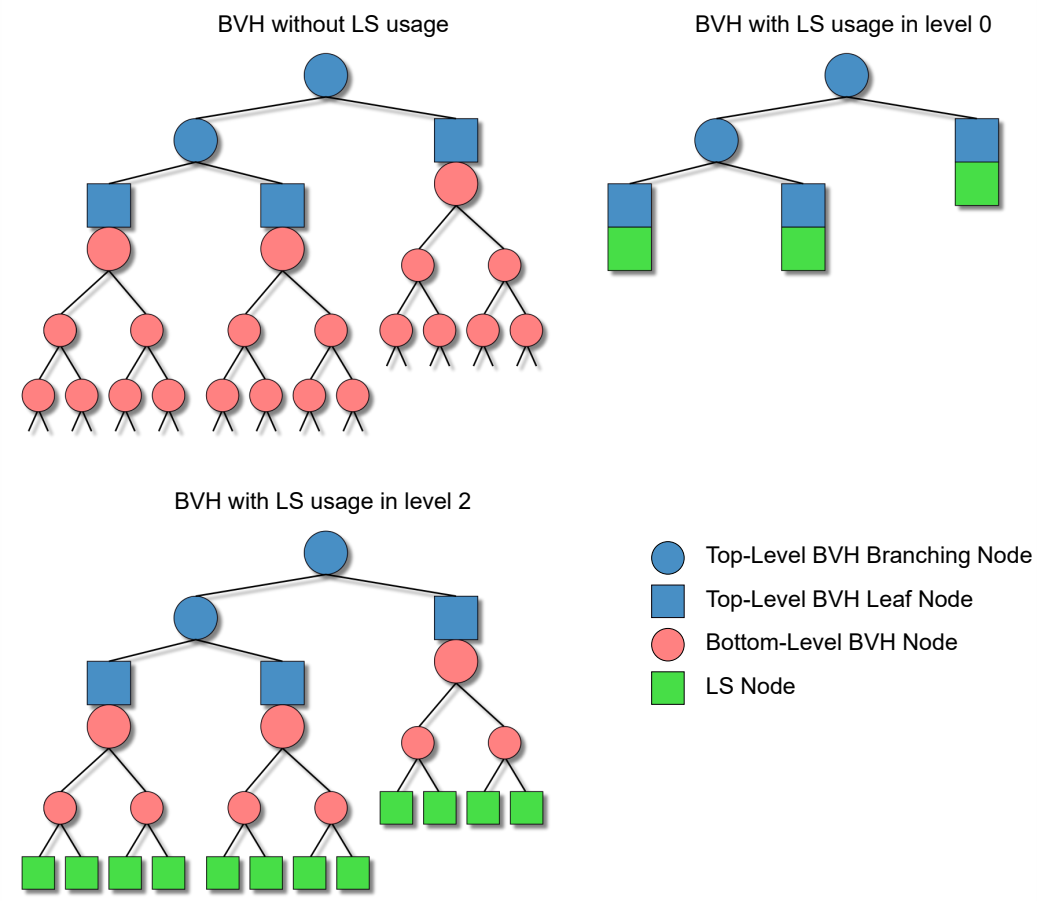


Figure 8.3: LS usage within different levels of bottom-level BVHs in a two-level hierarchy, as presented in chapter 7. Left: Pure two-level BVH without LS consideration, which results in correct occlusion determination. Middle: LS usage in level 2 of the bottom-level BVH, resulting in an initial BVH traversal, which is then terminated by LS approximations. Right: Instead of using the bottom-level BVH, only the LS approximations are used on object-level, which results in fast occlusion determination and visible artifacts. The work at hand dynamically decides, which level is used for LS integration based on the usage heuristic.

and reduced the needed memory. While this was only applied to indirect illumination and representative candidate approximation until now, it is fairly obvious that the LS incorporation within two-level structures is also beneficial for reducing shadow approximation artifacts, which is shown in the results. This can be also illustrated by abstraction of the tracing cost c of a spatial data structure like the BVH. This cost is in general the sum of the data structure's traversal cost c_t and the cost of the intersection tests c_i with all necessary scene primitives $\#p$:

$$c_{BVH} = c_t + c_i * \#p \tag{8.1}$$

LS approximation reduces this in two ways. First, the traversal is fixed to a specific depth of the LS integration d, due to it's early ray termination. In practice, LS integration in a depth $d <= 10$ is sufficient, while the total BVH hierarchy depth is oftentimes significantly higher ($>= 20$). Second, during ray termination a special attribute of the surrounding shaft is used for the ray. In shadow computation it is assigned as blocking or non-blocking; in indirect lighting computation the representative shaft candidate is assigned to the ray. In both cases, no further intersection tests are needed and therefore $c_i * \#p$ is not needed anymore. However, the cost of the LS access c_a is additionally needed, which is in the same order of magnitude as a single intersection test cost c_i:

$$c_{LS} = c_{t<d} + c_a \tag{8.2}$$

For two-level structures, the top-level data structure needs to be additionally considered. The cost c then applies to all leaf nodes of the top-level hierarchy; it is not affected by the LS usage and due to simplicity omitted in Equation 8.1 and Equation 8.2. The lower hierarchical depth trough LS usage for two-level structures is illustrated in Figure 8.3.

The LS initialization with binary visibility information can be done in two ways: through masking or ray casting. The former was used in chapter 4 and works with a simple idea. It is easily computable which voxels of the bounding box are intersected by a given shaft. By assigning all connections of shafts to voxels, this calculation can be reversed and it is visible which shafts are occupied by a specific voxel. All those shafts are stored in a voxel shaft mask. Doing so for every possible mask results in a mask atlas, which can be used to quickly identify all occupied shafts from a voxelized scene primitive or object. However, as the voxels can be much bigger than the contained scene elements, this is a vast overestimation and shafts are wrongly classified as non-empty. The second initialization approach is based on ray casting, as illustrated in chapter 6. During construction the midpoint ray for every shaft is created by the midpoints of the

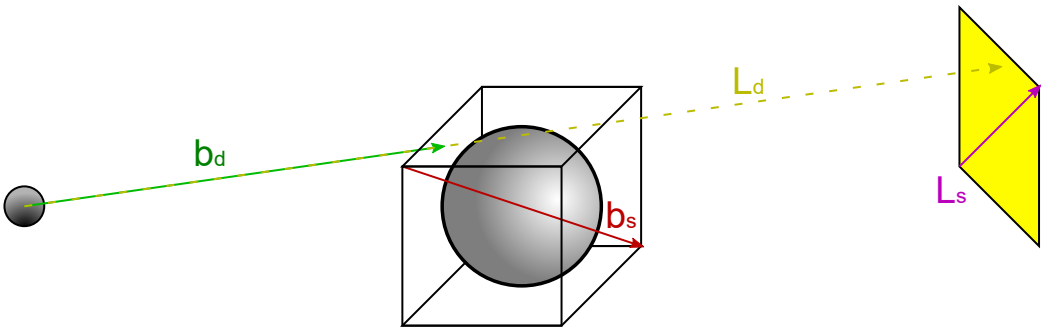


Figure 8.4: Illustration of the usage heuristic used in soft shadow calculations. Approximations are more critical when the object is bigger and near to the shadowed point. This is only measured based on the object's bounding box (with distance b_d and size b_s). Additionally, approximations are more crucial when the light source is smaller (L_s) or further away from the given point (L_d). This is summarized in the usage heuristic (Equation 8.3).

forming bounding box patches. This ray is then traced with the underlying spatial data structure. If an intersection is found, it is stored within the shaft, otherwise, the shaft is classified as empty. In summary, masking has lower initialization time and ray casting a better estimation and more precision. In this case, the LS is only initialized once per object, focusing more on precise shadow approximation and therefore ray casting is used for initialization.

8.3.2 Adaptive Combination

With this, there now exist two ways in computing shadows. The accurate way uses typical BVH ray tracing and achieves correct rendering results. Consequently, the efficient way uses LS approximations within BVH tracing, which enables the early ray termination and therefore has higher tracing performance, however at the cost of approximation artifacts. The idea is to combine both approaches and to use a simple heuristic that decides whether to use the LS shadow approximation. An adaptive approach was already used in a path tracing system, where the decision of using LS acceleration was simply done based on the current recursive path depth (chapter 7). The here presented usage heuristic is more elaborate in this and just as simple in terms of computation. It can be observed, that soft shadow approximations are less critical, when the blocking object is far away from the possibly shadowed point or if the blocking object is rather small in size. Additionally, the light source's distance and size is important

for approximation accuracy. If the ratio between the light source's size and it's distance is big, shadow approximations are less critical. In turn, shadow approximations are more crucial when the light source is far away from the shadowed point or small in size. This is especially visible when point light sources are used instead of area light sources. These observations are used in the usage heuristic, as illustrated in Figure 8.4:

$$\gamma = (\frac{b_d}{b_s} * \frac{L_s}{L_d}) * \delta \tag{8.3}$$

For simplicity the object's bounding box is used within the usage heuristic, with the box distance b_d and size b_s, along with the light's distance L_d and size L_s. The value of γ represents the distance of the bounding box in ratio to it's size. If this exceeds a given threshold, it indicates that LS approximations are good to use, while low values show that the accurate results of the BVH should be used.

$$\text{use} \begin{cases} \text{LS}, & \text{if } \gamma \geq 1 \\ \text{BVH}, & \text{otherwise} \end{cases} \tag{8.4}$$

The value of δ can be used to control this determination, so that it is possible to adjust the performance and quality. A higher value of δ results in more LS usage and higher performance, while a low value of δ produces more accurate tracing.

A new constructed shadow ray usually starts within the bounding box of the possibly shadowed point, first searching for self-shadowing. This is typically not possible to achieve with LS approximations, as the shaft always indicates non-emptiness and therefore that the ray is blocked by an object (which is itself). In previous work, the first bounding box was ignored due to this, leading to visible image errors. The presented heuristic always uses accurate BVH results for self-shadowing and therefore is able to fix these errors.

It is possible to use this heuristic not only to indicate if the LS is used, but also to decide in which level of the bottom-level BVH it is used. A higher value of γ indicates earlier LS usage, i.e. a lower BVH level. By subtracting the value of γ from the maximum LS level that is present, it is calculated which actual LS level is used for the ray traversal. This adaptive usage requires that all nodes up until the maximum level in the BVH contain their respective LS. Additionally, it is possible to interpolate the value of γ so that a smooth transition between different LS levels is achieved. This is illustrated in Figure 8.5.

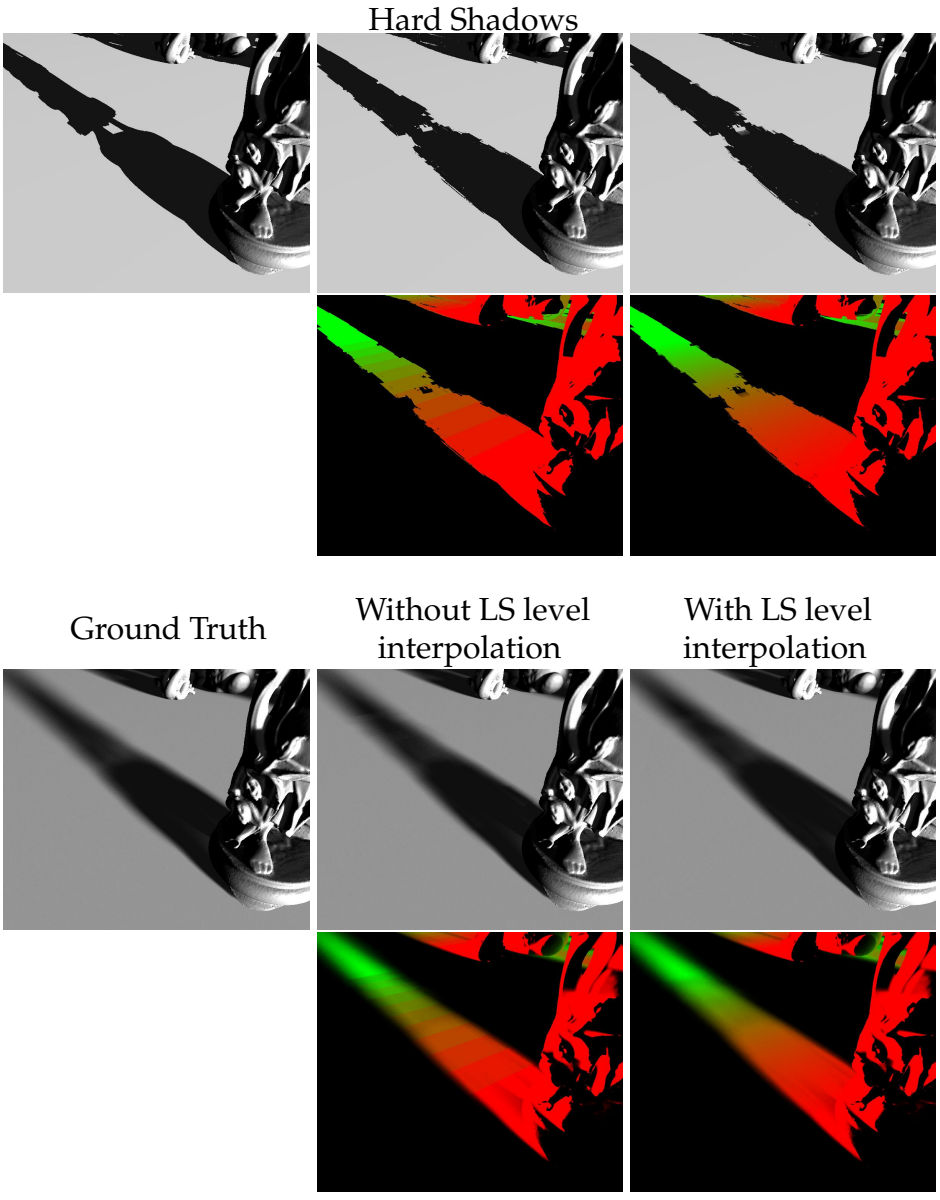


Figure 8.5: Illustration of the usage heuristic and it's application in adaptive determination whether to use correct BVH or fast approximate LS traversal. Red areas in the usage heuristic images mark low values of γ and therefore the need for correct traversal. Accordingly, green areas represent high values of γ and hence areas where it is sufficient to use LS approximations. Furthermore, the interpolation process between different γ values and thus different LS levels are presented. Obviously, LS approximations are less critical when used with area lights in order to produce soft shadows.

Algorithm 8 The top-level data structure traversal code for blocker determination in shadow calculation.

```
procedure TRACETOPLEVEL(Ray r, Light l)
    stack.push(root)
    while node ← stack.pop() ≠ 0 do
        if node.isLeaf() then
            γ ← δ * distance(r, node.aabb) / size(node.aabb) * size(l) / distance(r, l)
            if γ ≥ 1 then
                x ← maxLevel − γ
                shadow ← trace BVH with LS on level x
            else
                shadow ← trace pure BVH without LS
            end if
            if shadow then
                return true
            end if
        else
            add intersecting subnodes to stack
        end if
    end while
    return false
end procedure
```

The presented heuristic can be integrated in the tracing step of the top-level data structure, which can be traced as usual. However, when the top-level structure indicates that an object data structure should be traced, it is first checked with the usage heuristic whether accurate or high-performance tracing should be used. This traversal is shown in Algorithm 8.

8.3.3 Disc Sampling for Ambient Occlusion

Ambient occlusion can be calculated by tracing multiple rays starting from a given point with different directions along the half space aligned with the point's surface normal. The resulting average occlusion describes the possible light that is able to reach the given point. In other words, the sample rays are tested for blocker determination in order to receive an adequate approximation of the occlusion value. By generalization of rays to shafts, the LS actually simplifies this to fewer samples that are also easier to trace. Due to the averaging of an amount of rays, the actual scene geometry is less relevant and therefore LS approximations are less

critical. However, the LS produces visible artifacts at the bounding box's boundaries. When the shafts of a nearby LS node are used for blocker determination only a small amount of different shafts are actually sampled, while most rays are stuck in the same shafts over and over. This falsifies the ambient occlusion value at the boundaries of different LS nodes. This is fixed by using not the actual point as starting point for the ray, but different ones that are sampled on a disc aligned with the given point's surface normal[1]. By using a variance of different starting points, it is ensured that a greater variety of shafts are traced and therefore the falsification of ambient occlusion values disappears, as demonstrated in Figure 8.6.

For the disc sampling itself a uniform sampling is used through random angle and random distance from center point distribution. By this, more sampling points are in the center of the disc and therefore a smooth progression is achieved. The disc radius is set according to the value of γ of the usage heuristic (without using the light source parameters). If the disc radius is too low, it achieves nearly no result; if the value is too high, it smoothens ambient occlusion too much and relevant details become lost. By again involving the usage heuristic, it is able to limit the disc sampling to sections where it is appropriate.

8.3.4 Line Space Parameter Discussion

As in previous work, two main parameters for the LS are identified, the subdivision parameter N and the integration depth d within the object BVH. While previous work on acceleration for soft shadows through LS usage worked on the complete scene in a global structure, here the focus is on local LS on object level through the two-level approach. With this, it is able to rely on a fixed LS integration level within the base hierarchy, which limits the total amount of generated LS nodes and therefore the needed memory. As in previous work on two-level LS, the object structures are only computed once per mesh and reused when the mesh occurs multiple times within the scene.

The memory consumption of a single LS node is only dependent from the number of shafts, which are determined by the subdivision parameter of the bounding box, resulting in $15N^4$ shafts for a single LS node, similar to previous work. Every shaft consists of one bit of information, indicating whether it is empty or non-empty. The number of necessary LS nodes only

[1]Thanks to Maximilian Nilles for productive discussion on the disc sampling strategy. For different application of the disc sampling, it is referred to his thesis "Light-Injection und Global Illumination mittels GPU und der Linespace Datenstruktur" (2017).

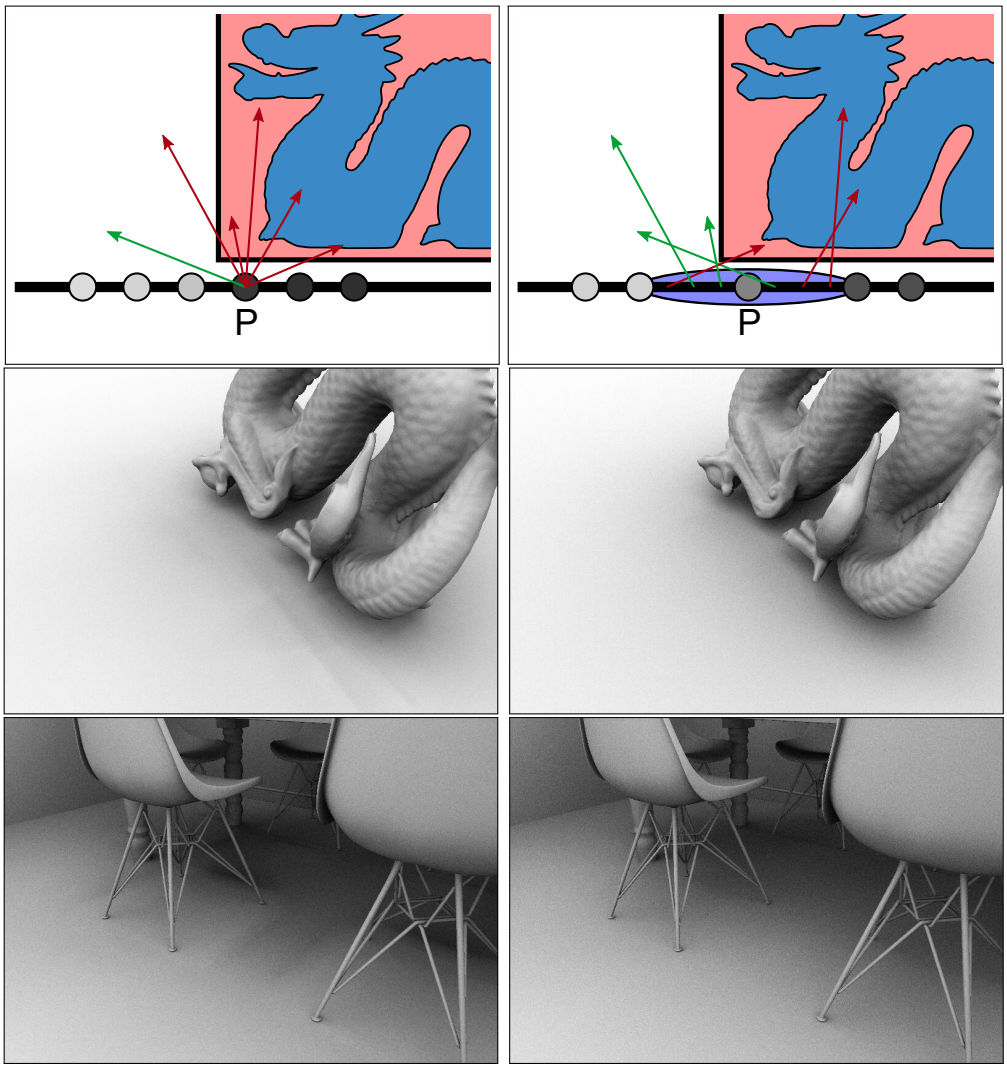


Figure 8.6: The usage of disc sampling during ambient occlusion calculation significantly reduces LS artifacts that appear due to shaft approximation and thereby systematically smoothens the transition of LS edges. This is illustrated in the top images, where P is a sample point without and with disc sampling. The middle and bottom images also show, that the actual degree of artifacts is scene dependent.

MB	d=0	d=2	d=4	d=6	d=8	d=10
N=4	<0.01	<0.01	0.01	0.06	0.23	0.94
N=6	<0.01	0.02	0.07	0.29	1.18	4.74
N=8	0.01	0.05	0.23	0.93	3.74	14.99
N=10	0.02	0.13	0.55	2.27	9.14	36.6

Table 8.1: The memory consumption (in MB) per geometrical mesh for different values of the subdivision parameter N and the integration depth d. The memory size does not depend on the actual geometry.

depends on the integration depth d within the object BVH. The needed memory size for the LS is therefore independent from the actual scene geometry. Some exemplary values are presented in Table 8.1. It is visible, that high values for N and d lead to much higher memory consumption. However, thanks to the system's adaptive design and the incorporation of the usage heuristic, it is possible to benefit from low parameter values and therefore have low memory consumption. This is further beneficial in terms of runtime performance, as a low depth parameter for LS integration results in earlier ray termination and hence faster tracing times.

8.4 Results

The adaptive structure is implemented with the usage heuristic and disc sampling for ambient occlusion using GLSL Compute Shaders on a testing system consisting of a NVidia GeForce GTX 1080 GPU and an Intel i7-6800k 3.6GHz CPU with 16GB RAM. The adaptive LS structure is integrated within a state-of-the-art two-level BVH approach, which is also used for ground truth comparison. A BVH construction algorithm targeting good tree quality with non-interactive build times is used for both, the top-level as well as the object BVHs. However, as the LS data structure is independent from the base structure and can be integrated within any structure that is based on bounding boxes in any kind, other construction algorithms or even base data structures work as well for the object structures. For BVH traversal the algorithm presented by [Aila et al., 2012] is used and adjusted to the adaptive design. The traversal of the top-level structure is based on Algorithm 8. Based on the presented usage heuristic, the object structure traversal either uses typical BVH tracing and thereby generates accurate shadow results or uses early ray termination by the LS in a given tree depth d, which generates fast approximated blocker determination. The two-level

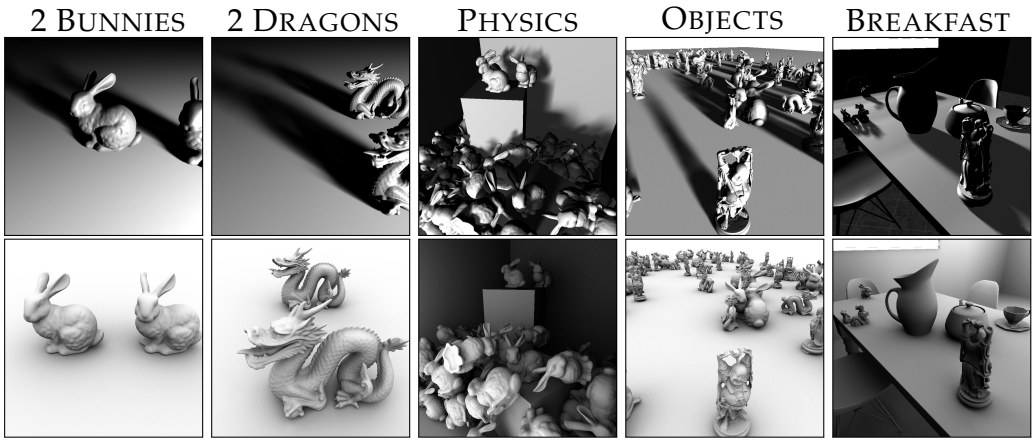


Figure 8.7: The test scenes used for the evaluation with soft shadows (top row) and ambient occlusion (bottom row). The 2 BUNNIES and 2 DRAGONS scenes use individual objects on top of a plane. The PHYSICS and OBJECTS scenes heavily rely on object instancing, as used in simulation. The BREAKFAST scene is an architectural scene that contains single objects, as for example used in video games.

BVH is compared with and without the LS usage in terms of memory consumption, tracing performance and image error in shadow and ambient occlusion generation. In total, it is shown that, thanks to the usage heuristic, the adaptive approach produces nearly non-visible approximation errors while also offering a significant boost in tracing performance in comparison to traditional two-level BVHs.

The chosen test scenes can be categorized in different scenarios. It is started by using single models on top of a plane, in order to indicate the quality gain produced by the two-level approach. Then, the additional potential in terms of object instancing and rigid object transformations through the two-level structure is used. Therefore, test scenes with a high amount of instanced objects are produced, where the object hierarchy is only generated once per geometric mesh. Finally, these single model meshes are combined with architectural scenes and hence scenes are involved that are for example typically used in video games. The used test scenes are visualized in Figure 8.7.

Memory Consumption

In previous work the high memory consumption was one of the LS's main limitations. However, in the current approach this is not the case due to some reasons. As presented in Table 8.1 only a small amount of memory

Mesh	Cube	Bunny	Dragon	Buddha
# triangles	12	69k	871k	1087k
BVH depth	4	19	26	26
# BVH nodes	7	44k	578k	719k
BVH size (MB)	<0.1	1.7	22.0	27.4
Line Space with $N = 6$ and maximum depth $d = 6$				
# LS nodes	7	127	127	127
LS size (MB)	<0.1	0.3	0.3	0.3
LS init time (s)	0.019	0.278	2.820	3.461
Line Space with $N = 6$ and maximum depth $d = 10$				
# LS nodes	7	2047	2047	2047
LS size (MB)	<0.1	4.7	4.7	4.7
LS init time (s)	0.019	0.280	2.871	3.538

Table 8.2: Different characteristics of geometrical meshes used as bottom-level objects in the two-level structure. Obviously the BVH depth and it's memory size scale with the number of object triangles. The LS is integrated up until a maximum depth given by d and therefore only a limited number of LS nodes are generated, independently from the object itself.

is needed per geometrical mesh for parameter sets with low values of the subdivision parameter N and integration depth parameter d. In all tests a value of 6 is used for N, which is in accordance to previous work and was sufficient for image quality. Thanks to the two-level approach, the maximum integration depth can be fixed and therefore only a small calculable number of LS nodes is generated per mesh, in contrast to previous results. Furthermore, as in previous work on LS produced soft shadows, only a single bit is used for emptiness identification per shaft, which makes the presented structure extremely memory efficient and the shaft abstraction makes LS blocker determination independent from contained scene geometry. Especially when used with complex meshes and thus high BVH depths, the needed memory size of the LS is much lower compared to the underlying BVH. The results in terms of memory consumption and initialization speed with two different maximum integration depths d are presented in Table 8.2. By working on two-level hierarchies, the overall memory consumption of the data structure is mainly influenced by those bottom-level structures. The top-level BVH structure then only refers to these bottom-level structures, which is especially beneficial in scenes that

Scene	2 BUNNIES	2 DRAGONS	PHYSICS	OBJECTS	BREAKFAST
# different meshes	2	2	3	4	48
# instances (incl. floor)	3	3	67	151	50
Top-Level BVH depth	3	3	9	11	10
# virtual triangles	139k	1742k	4467k	101300k	4040k
Soft Shadows					
BVH Perf. (MRays/s)	320.1	139.2	103.2	48.5	76.9
Ad. LS Perf. (MRays/s)	521.7	212.3	168.4	118.9	114.3
Improvement	x1.63	x1.53	x1.63	x2.45	x1.49
Ad. LS Error (RMSE)	0.005	0.006	0.014	0.018	0.005
Ambient Occlusion					
BVH Perf. (MRays/s)	198.7	106.6	65.3	50.8	36.4
Ad. LS Perf. (MRays/s)	355.7	168.2	106.1	84.2	68.3
Improvement	x1.79	x1.58	x1.62	x1.66	x1.88
Ad. LS Error (RMSE)	0.013	0.016	0.017	0.015	0.021

Table 8.3: The test results for the different scenes presented in Figure 8.7. The number of virtual triangles describes the total number of scene triangles, covering all object instances. The adaptive LS usage is especially beneficial, if many complex objects with a high number of triangles are used in the scene. Furthermore it is visible, that the performance (given in million rays per second, MRays/s) is in all cases significantly higher by using the LS and the produced approximation error (given in root-mean-square-error per pixel, RMSE) stays quite low.

heavily rely on object instancing, as the geometrical mesh object itself and thus the LS and BVH only need to be stored once.

Performance

The tracing performance is measured for shadow and ambient occlusion computation per scene in million rays per second (MRays/s) and is therefore independent from screen resolution. Obviously, the additional LS usage as termination criterion in occlusion approximation accelerates tracing performance significantly. As illustrated in Table 8.3, an average speedup factor of around 1.7× with a peak of up to 2.5× can be observed for all scenes. The results demonstrate, that the greatest acceleration is in scenes with a big number of high-resolution meshes. There, the LS is able

to reduce the expensive bottom-level BVH traversal and intersection tests in most cases with the fast shaft-based termination. In general, the LS is more beneficial with a higher number of complex occluding objects, but even in the worst case, the rendering performance is similar to typical BVH traversal, as this is the underlying base data structure.

Approximation quality

The incorporation of LS approximation typically leads to a loss in quality and visible artifacts, especially when low parameter values are used. Previous work reduced these errors by using a two-level hierarchy, which is also used in the work at hand. However, a significant part of artifacts remained, which are in particular visible at the object surface and near to it. These errors are eliminated with the adaptive technique thanks to the usage heuristic. By this, the remaining error is nearly non-visible, as demonstrated in Figure 8.8. The results are compared with correct BVH renderings as ground truth and measure the error with a per pixel root-mean-square-error (RMSE), which is presented in Table 8.3. It is visible that for both rendering techniques, soft shadows as well as ambient occlusion, the errors due to approximation are quite low. In case of shadows, the resulting error becomes more critical when the light source is small in size or near to the occluding object, illustrated in Figure 8.5. However, the incorporation of the usage heuristic especially handles those cases and there leads to increased utilization of correct results at the cost of runtime performance. By this, the resulting error remains small, and even in extreme scenarios, where the correct BVH traversal is used nearly always, the performance is at least as fast as state-of-the-art, as explained above.

8.5 Conclusion and Future Work

With this work an adaptive technique is proposed, where two-level BVHs are combined with precomputed LS approximations through the utilization of a simple yet efficient usage heuristic that decides on a per object level in the two-level structure, whether it is beneficial to use approximated fast LS results in blocker determination computation. With this adaptive approach it is possible to eliminate a significant amount of approximation errors in soft shadow generation of previous work while also maintaining good traversal acceleration compared to typically used spatial data structures. Furthermore, a way to use this approach for ambient occlusion calculation through a disc sampling strategy is presented. By this, a trend is continued,

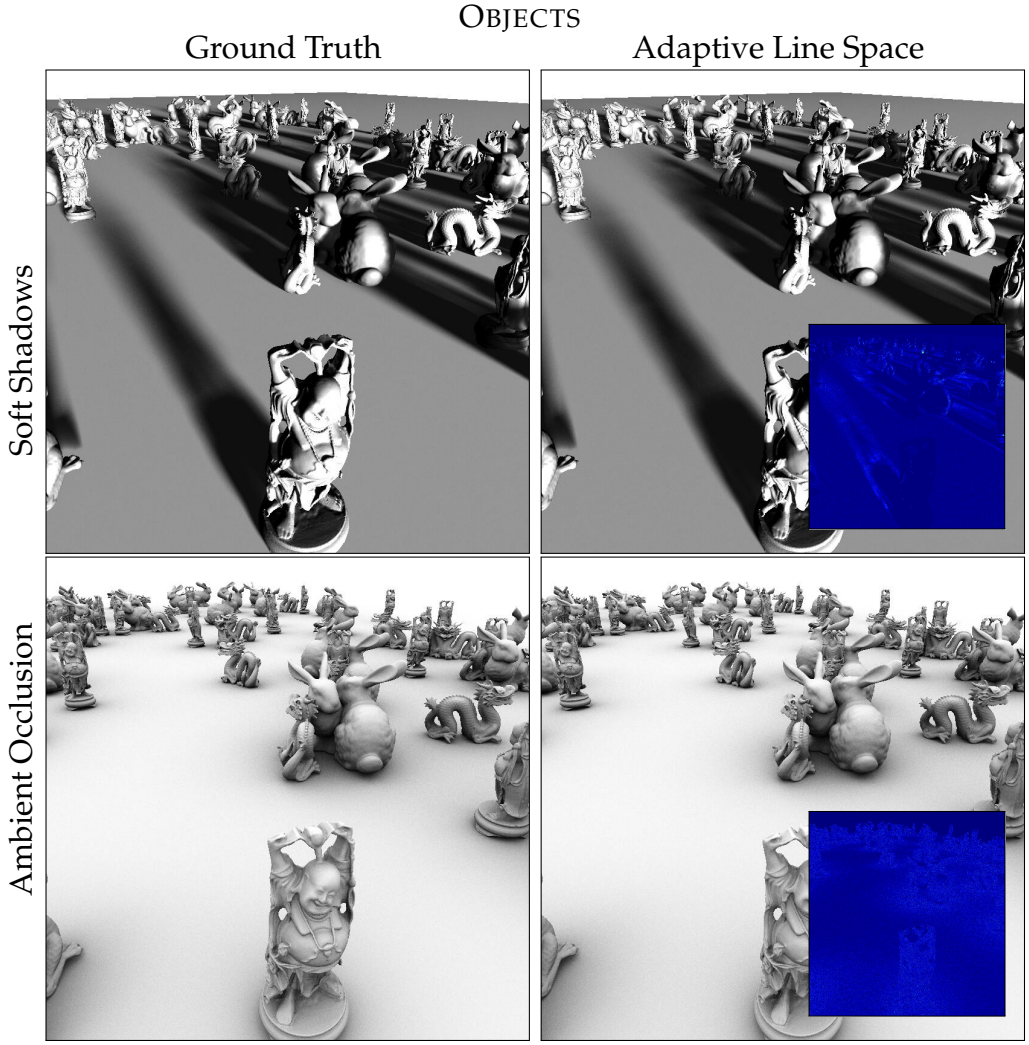


Figure 8.8: Soft shadow and ambient occlusion results for the OBJECTS scene. It is visible, that LS approximations are nearly non-visible thanks to the usage heuristic. Furthermore, the LS incorporation significantly improves performance, as presented in Table 8.3.

where the incorporation of directional precomputed visibility information through shaft abstraction in the LS is able to further accelerate spatial data structure in ray tracing systems.

The main benefit of the LS integration within typical spatial data structures is the gain in performance through usage of precomputed approximations. Obviously, for future work the incorporation of the presented usage heuristic into indirect and global illumination computation through representative candidate precomputation based on shafts may result in a promising trade-off between performance and approximation quality. Until now, only the integration depth d is dependent from the usage heuristic; therefore, the next step is to additionally use a dynamic value of the branching parameter N for better trade-off. Finally, the precomputation of shaft information may be further investigated and applied to approximation based volume rendering acceleration.

CHAPTER 9

CONCLUSION

This chapter summarizes the results and achievements of this work and thereby presents a conclusion of all contributions made. Based on this, different ideas for future work are motivated and it is illustrated in which way the results of this work may be beneficial for further progress in ray tracing acceleration and beyond.

9.1 Summary

As stated at the beginning of this work, the main vision is to find a suitable structure for precomputed information that can be used to determine the visibility of a given point in a given direction. As the complete precomputation is obviously too immense in terms of memory consumption and initialization time for non-trivial scenes and current hardware, it is necessary to find an efficient abstraction, such that precomputed directional visibility information can be stored and used effectively. This work presents the LS as such a structure, based on shafts that store the visibility information.

The main objective of this thesis is the presentation and discussion of this new approach as a combined approach between typically used spatial data structures and precomputed directional visibility information. For this purpose, the current state-of-the-art in ray tracing acceleration through spatial and visibility structures is presented in chapter 2. Afterwards, the LS is proposed as some kind of classification method for rays, which are traversed with the spatial acceleration structure as base structure. In chapter 3, it is illustrated in which way the LS is an extension to previous work of ray classification. More than that, it is shown how precomputed visibility information is contained within the abstraction of shafts and how it is integrated into the spatial base structure. Additionally, the main

concepts and properties of the LS are presented and set into perspective for the following chapters.

In chapter 4, it is then described how the LS is filled with binary visibility information, indicating whether a shaft is empty or non-empty. This is used for a simple yet efficient empty space skipping technique. In that chapter, the LS is integrated into the *N*-tree, a recursive grid structure, which builds on the same subdivision strategy as the LS. Furthermore, a fast construction algorithm based on masking is presented. Consequently, it is shown that the additional usage of the LS achieves a performance gain of up to 1.4× compared to the pure *N*-tree in a CPU ray tracing system for all rays independently of their ray type. Thus, the LS effectively trades an amount of memory space for runtime performance.

The same premise is used differently in the approximation of soft shadows in chapter 5. While the binary shaft information was previously used as an indication to skip further investigation of empty areas during ray traversal, it is then used in the opposite way for shadow generation, as non-empty shafts indicate volume that contains potentially blocking objects. This criterion is used as blocker determination instead of the correct intersection tests, leading to approximation errors and therefore inaccurate but fast shadow calculation. To reduce the perception of image errors, the technique is used for soft shadows from area lights instead of hard shadows from point lights. Additionally, the ray tracing process and therefore the data structure initialization and organization are computed on the GPU. Overall, it is shown that a remarkable performance gain of up to 3× with only a low amount of image errors is achieved by the additional LS usage.

Based on these findings, the type of visibility information stored in the LS shafts is extended in chapter 6 to a single candidate that is used as representative for all rays that pass the shaft. Thereby, the needed memory increased significantly and thus an efficient memory layout is proposed. Furthermore, the LS usage is extended in order to work based on bounding boxes and therefore enables the integration into state-of-the-art BVHs. To initialize the LS structure with the representative candidate, an initialization technique based on ray tracing is presented. The representative is then used during runtime to approximate the actual surface intersected by a ray passing the shaft. For all rays that are not intersected by the representative, an algorithm based on extrapolation with a safety condition based on the surface normal is proposed in order to limit the image errors due to approximation. As before, the technique results in approximations and image errors, however, as it is used only for indirect lighting, this error is nearly not perceivable. Again, the additional LS usage results in a performance gain of up to one order of magnitude for integration in the

N-tree and of up to $3\times$ when integrated into state-of-the-art BVHs, which is due to the limitation of the ray's hierarchy traversal and the immense reduction of needed intersection tests.

In order to limit the previous image errors of indirect illumination approximation, an adaption to two-level structures is presented in chapter 7. There, the LS is only used in the object level, resulting in much more precise LS approximations, however at the cost of runtime performance, as two-level structures generally have lower tracing performance. Apart from this, the adaption to two-level structures enables the possibility of rigid object transformations and instancing, even with precomputed LS visibility information, and thus decreases the needed memory and construction time. In scenes that heavily rely on object instancing, this furthermore presents a gain in runtime performance. Concerning LS usage, this adaption overall grants more opportunities for parameter improvement, which finally results in an adaptive path tracing technique, where different LS structures are used in combination depending on the current path depth. In total, this results in a better trade-off between runtime performance and image errors for different stages of path tracing.

The previous adaption to two-level structures also enables the opportunity for a more advanced determination whether fast LS approximations or correct results are used in traversal. This leads to the description of a usage heuristic in chapter 8. There, the LS is integrated into different depths on object level up to a given depth. During traversal of the top-level data structure, the usage heuristic is utilized in order to decide if LS approximations are sufficient and which integration depth is used for occlusion determination. By this, it is also possible to use correct results where they are needed, which leads to a significant reduction of approximation errors in the results. The usage heuristic is furthermore utilized in a disc sampling technique which reduces artifacts of LS involvement in ambient occlusion acceleration. Finally, it is shown that the incorporation of the usage heuristic significantly reduces approximation errors in LS occlusion determination while at the same time maintaining most of the gain in tracing performance.

Overall, the LS is presented as a practical next step towards complete visibility precomputation. Through the integration into traditional and state-of-the-art spatial data structures, it is possible to reduce the required memory and initialization time to a reasonable extent. Though some errors due to approximation are included, different techniques are presented to reduce these errors to a nearly non-visible amount. Therefore, the LS is a viable novel structure for enhanced ray tracing acceleration with big potential for future work.

9.2 Future Work

All results used in this work are used with the objective of accelerating ray tracing in rendering. Future work can benefit in multiple ways from this. Some possibilities are proposed in the following, starting with extensions and advancements close to the topics of this thesis and finally leading to a wider context in which LS precomputations may be used.

First of all, the LS as presented here is based on parameter sets in a given range, which were experimentally shown to give good results in the given circumstances. The parameter values are chosen based on the results and are fixed for the complete data structure. An adaptive scheme may give better results and it is therein possible that the subdivision resolution can be chosen depending on the content of the current node. This may result in higher runtime performance and more efficient memory consumption, but needs further indication in deciding which parameters are optimal for given geometrical contents based on their primitive amount, size and orientation. A simple yet effective heuristic may use the number of contained scene primitives in a given node as an indication for a good subdivision parameter. Moreover, this may also lead to more sophisticated construction algorithms of the spatial base data structure, as the subdivision parameter and the integration depth may also affect the subdivision scheme of the underlying spatial structure. In that, this might lead to a more elaborate variation of the surface area heuristic (SAH), which may be used for data structures that build on LS precomputations.

For now, the LS initialization takes quite some time. While this is only in a matter of seconds, it is not able to recompute the structure during runtime. A few promising ideas that may change this are quite obvious. The construction is currently done through masking or ray casting in the initialization step. However, during rasterization it is possible to compute all shafts that lie in direction of a given point, i.e. the camera or light sources. Using this, it may also be possible to use a rasterization-based approach for initialization, instead of ray casting. This may significantly improve construction performance and, when only used incrementally for shafts that are actually visible from the current camera position, this may lead to interactive construction times and further usage in real-time rendering.

Apart from that, the type of precomputed visibility information within shafts is one of the most promising features of the LS. In previous work, similar structures used the complete list of scene candidates in each beam for ray classification. This thesis involves shafts instead of beams and uses binary visibility information indicating the emptiness and furthermore a representative candidate per shaft. These strategies are shown to result in

good ray tracing acceleration by reduction of the needed intersection tests and incorporation of approximations in soft shadow and indirect lighting computation. More elaborate visibility precomputation schemes may also involve different other information. Especially concerning rendering, there are some obvious opportunities. Apart from precomputation of blocking scene candidates, the actual final lighted color of scene primitives with respect to their material can be precomputed for shafts, which may already involve global illumination. This would trade even longer initialization time for fast high-quality color calculations without further computation overhead, nevertheless, additional investigation in rasterization-based construction may reduce the initialization time. However, this has some significant downsides, which have to be solved first. The main limitation is the approximation error due to shaft abstraction. For images with reasonably good quality, a high LS resolution is needed, which in turn results in tremendous amounts of necessary memory and long initialization times. Apart from that, the second limitation is the actual value of the usability of such a structure. It aims towards real-time calculations, however, disables the possibility of dynamically moving scene objects or changing lighting settings, as those require recalculation of the data structure, which is not yet possible to achieve in real-time. As a simplification to this concept, it is possible to only precompute lighting information, instead of the final lighted and shaded color. The main idea may be similar to the concept of radiosity, but due to the regular subdivision of the LS nodes, this may be faster in most cases. Concerning the big vision of complete visibility precomputation, this computation and storage of the actual lighting per shaft may in fact be the next big step.

While all discussed methods aim towards the acceleration of global diffuse illumination, it may also be possible to use the LS precomputation for different scenarios, such as volume rendering. In that, it is mostly necessary to have some information about the participated volume's properties, which can be stored within the shafts. For this, it is possible to store all directional information of the shafts during initialization, such as all entry and exit points of the scene geometry and the material properties of the passed volume. During runtime, it is then only necessary to use the already calculated information of the LS instead of actually searching for intersection points with the volume, which may as well significantly improve rendering performance.

Lastly, it is pointed out that directional precomputations through the LS may be used in completely different areas. An example of this may be acoustic wave propagation, where the directional distribution of sound is

measured. This is currently mostly done with some kind of ray tracing, and LS precomputations might also accelerate such computations.

Own Publications

[Keul et al., 2017] Keul, K., Klee, N., and Müller, S. (2017). Soft shadow computation using precomputed line space visibility information. *Journal of International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG)*, 25:97–106.

[Keul et al., 2018] Keul, K., Koß, T., and Müller, S. (2018). Fast indirect lighting approximations using the representative candidate line space. *Journal of International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG)*, 26:11–20.

[Keul et al., 2019] Keul, K., Koß, T., Schröder, F. L., and Müller, S. (2019). Combining two-level data structures and line space precomputations to accelerate indirect illumination. In *Computer Vision, Imaging and Computer Graphics Theory and Applications*.

[Keul et al., 2016] Keul, K., Müller, S., and Lemke, P. (2016). Accelerating spatial data structures in ray tracing through precomputed line space visibility. In *Proceedings International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG)*, volume 24, pages 17–25.

BIBLIOGRAPHY

[Áfra and Szirmay-Kalos, 2014] Áfra, A. T. and Szirmay-Kalos, L. (2014). Stackless multi-bvh traversal for cpu, mic and gpu ray tracing. In *Computer Graphics Forum*, volume 33, pages 129–140. Wiley Online Library.

[Aila et al., 2013] Aila, T., Karras, T., and Laine, S. (2013). On quality metrics of bounding volume hierarchies. In *Proceedings High Performance Graphics (HPG)*, pages 101–107. ACM.

[Aila and Laine, 2009] Aila, T. and Laine, S. (2009). Understanding the efficiency of ray traversal on gpus. In *Proceedings High Performance Graphics (HPG)*, pages 145–149. ACM.

[Aila et al., 2012] Aila, T., Laine, S., and Karras, T. (2012). Understanding the efficiency of ray traversal on gpus–kepler and fermi addendum. *NVIDIA Technical Report*.

[Akenine-Möller et al., 2008] Akenine-Möller, T., Haines, E., and Hoffman, N. (2008). *Real-time rendering*. CRC Press.

[Amanatides, 1984] Amanatides, J. (1984). Ray tracing with cones. In *ACM Siggraph Computer Graphics*, volume 18, pages 129–135.

[Amanatides et al., 1987] Amanatides, J., Woo, A., et al. (1987). A fast voxel traversal algorithm for ray tracing. In *Eurographics*, volume 87, pages 3–10.

[Apetrei, 2014] Apetrei, C. (2014). Fast and Simple Agglomerative LBVH Construction. In *Computer Graphics and Visual Computing (CGVC)*. The Eurographics Association.

[Arvo and Kirk, 1987] Arvo, J. and Kirk, D. (1987). Fast ray tracing by ray classification. In *ACM Siggraph Computer Graphics*, volume 21, pages 55–64.

[Assarsson and Akenine-Möller, 2003] Assarsson, U. and Akenine-Möller, T. (2003). A geometry-based soft shadow volume algorithm using graphics hardware. *ACM Transactions on Graphics (TOG)*, 22:511–520.

[Barringer and Akenine-Möller, 2014] Barringer, R. and Akenine-Möller, T. (2014). Dynamic ray stream traversal. *ACM Transactions on Graphics (TOG)*, 33:151.

[Benthin et al., 2006] Benthin, C., Wald, I., Scherbaum, M., and Friedrich, H. (2006). Ray tracing on the cell processor. In *IEEE Symposium on Interactive Ray Tracing (IRT)*, pages 15–23.

[Benthin et al., 2012] Benthin, C., Wald, I., Woop, S., Ernst, M., and Mark, W. R. (2012). Combining single and packet-ray tracing for arbitrary ray distributions on the intel mic architecture. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 18:1438–1448.

[Benthin et al., 2017] Benthin, C., Woop, S., Wald, I., and Áfra, A. T. (2017). Improved two-level bvhs using partial re-braiding. In *Proceedings High Performance Graphics (HPG)*, page 7. ACM.

[Billen and Dutré, 2016] Billen, N. and Dutré (2016). Visibility acceleration using efficient ray classification. *Department of Computer Science, KU Leuven*.

[Binder and Keller, 2016] Binder, N. and Keller, A. (2016). Efficient stackless hierarchy traversal on gpus with backtracking in constant time. In *Proceedings High Performance Graphics (HPG)*, pages 41–50. Eurographics Association.

[Bittner et al., 2013] Bittner, J., Hapala, M., and Havran, V. (2013). Fast insertion-based optimization of bounding volume hierarchies. In *Computer Graphics Forum*, volume 32, pages 85–100. Wiley Online Library.

[Bittner et al., 2015] Bittner, J., Hapala, M., and Havran, V. (2015). Incremental bvh construction for ray tracing. *Computers & Graphics*, 47:135–144.

[Bittner et al., 2001] Bittner, J., Wonka, P., and Wimmer, M. (2001). Visibility preprocessing for urban scenes using line space subdivision.

In *Proceedings Computer Graphics and Applications*, pages 276–284. IEEE.

[Bresenham, 1965] Bresenham, J. E. (1965). Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4:25–30.

[Cohen and Wallace, 1993] Cohen, M. F. and Wallace, J. R. (1993). *Radiosity and realistic image synthesis*. Elsevier.

[Crassin and Green, 2012] Crassin, C. and Green, S. (2012). Octree-based sparse voxelization using the gpu hardware rasterizer. *OpenGL Insights*, pages 303–318.

[Crassin et al., 2009] Crassin, C., Neyret, F., Lefebvre, S., and Eisemann, E. (2009). Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Symposium on Interactive 3D Graphics and Games (I3D)*, pages 15–22, New York, NY, USA. ACM.

[Crassin et al., 2011] Crassin, C., Neyret, F., Sainz, M., Green, S., and Eisemann, E. (2011). Interactive indirect illumination using voxel cone tracing. In *Computer Graphics Forum*, volume 30, pages 1921–1930. Wiley Online Library.

[Crow, 1977] Crow, F. C. (1977). Shadow algorithms for computer graphics. In *ACM Siggraph Computer Graphics*, volume 11, pages 242–248.

[Dammertz et al., 2008] Dammertz, H., Hanika, J., and Keller, A. (2008). Shallow bounding volume hierarchies for fast simd ray tracing of incoherent rays. In *Computer Graphics Forum*, volume 27, pages 1225–1233. Wiley Online Library.

[Danilewski et al., 2010] Danilewski, P., Popov, S., and Slusallek, P. (2010). Binned sah kd-tree construction on a gpu. *Saarland University*, pages 1–15.

[Davidovič et al., 2014] Davidovič, T., Křivánek, J., Hašan, M., and Slusallek, P. (2014). Progressive light transport simulation on the gpu: Survey and improvements. *ACM Transactions on Graphics (TOG)*, 33:29:1–29:19.

[Domingues and Pedrini, 2015] Domingues, L. R. and Pedrini, H. (2015). Bounding volume hierarchy optimization through agglomerative treelet restructuring. In *Proceedings of the 7th Conference on High-Performance Graphics*, pages 13–20. ACM.

[Dong et al., 2004] Dong, Z., Chen, W., Bao, H., Zhang, H., and Peng, Q. (2004). Real-time voxelization for complex polygonal models. In *Proceedings Computer Graphics and Applications*, pages 43–50, Washington, DC, USA. IEEE Computer Society.

[Donnelly and Lauritzen, 2006] Donnelly, W. and Lauritzen, A. (2006). Variance shadow maps. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 161–165. ACM.

[Drettakis and Sillion, 1997] Drettakis, G. and Sillion, F. (1997). Interactive update of global illumination using a line-space hierarchy. In *ACM Siggraph Computer Graphics*, pages 57 – 64.

[Dutré et al., 1993] Dutré, Ph., Lafortune, E. P., and Willems, Y. D. (1993). Monte carlo light tracing with direct computation of pixel intensities. In *3rd International Conference on Computational Graphics and Visualisation Techniques*, pages 128–137, Alvor, Portugal.

[Eisemann and Décoret, 2006] Eisemann, E. and Décoret, X. (2006). Fast scene voxelization and applications. In *Symposium on Interactive 3D Graphics and Games (I3D)*, pages 71–78, Redwood City, United States. ACM Siggraph.

[Eisemann and Décoret, 2008] Eisemann, E. and Décoret, X. (2008). Single-pass gpu solid voxelization for real-time applications. In *Proceedings Graphics Interface*, pages 73–80, Toronto, Ont., Canada, Canada. Canadian Information Processing Society.

[Eisemann et al., 2011] Eisemann, E., Schwarz, M., Assarsson, U., and Wimmer, M. (2011). *Real-time shadows*. CRC Press.

[Ernst and Greiner, 2008] Ernst, M. and Greiner, G. (2008). Multi bounding volume hierarchies. In *IEEE Symposium on Interactive Ray Tracing (IRT)*, pages 35–40.

[Fang et al., 2000] Fang, S., Fang, S., Chen, H., and Chen, H. (2000). Hardware accelerated voxelization. *Computers & Graphics*, 24:433–442.

[Fernando, 2005] Fernando, R. (2005). Percentage-closer soft shadows. In *ACM Siggraph Computer Graphics*, page 35.

[Foley and Sugerman, 2005] Foley, T. and Sugerman, J. (2005). Kd-tree acceleration structures for a gpu raytracer. In *Proceedings ACM Siggraph/Eurographics Graphics hardware*, pages 15–22. ACM.

[Fuetterling et al., 2015] Fuetterling, V., Lojewski, C., Pfreundt, F.-J., and Ebert, A. (2015). Efficient ray tracing kernels for modern cpu architectures. *Journal of Computer Graphics Techniques (JCGT)*, 4.

[Fuetterling et al., 2016] Fuetterling, V., Lojewski, C., Pfreundt, F.-J., and Ebert, A. (2016). Parallel spatial splits in bounding volume hierarchies. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 21–30.

[Gaitatzes et al., 2010] Gaitatzes, A., Andreadis, A., Papaioannou, G., and Chrysanthou, Y. (2010). Fast approximate visibility on the gpu using precomputed 4d visibility fields. In *Proceedings International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision, (WSCG)*.

[Gaitatzes et al., 2008] Gaitatzes, A., Chrysanthou, Y., and Papaioannou, G. (2008). Presampled visibility for ambient occlusion. In *Proceedings International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision, (WSCG)*.

[Ganestam et al., 2015] Ganestam, P., Barringer, R., Doggett, M., and Akenine-Möller, T. (2015). Bonsai: rapid bounding volume hierarchy generation using mini trees. *Journal of Computer Graphics Techniques (JCGT)*, 4.

[Ganestam and Doggett, 2016] Ganestam, P. and Doggett, M. (2016). Sah guided spatial split partitioning for fast bvh construction. In *Computer Graphics Forum*, volume 35, pages 285–293. Wiley Online Library.

[Garanzha, 2008] Garanzha, K. (2008). Efficient clustered bvh update algorithm for highly-dynamic models. In *IEEE Symposium on Interactive Ray Tracing (IRT)*, pages 123–130.

[Garanzha et al., 2011] Garanzha, K., Pantaleoni, J., and McAllister, D. (2011). Simpler and faster hlbvh with work queues. In *Proceedings High Performance Graphics (HPG)*, pages 59–64. ACM.

[Georgiev et al., 2012] Georgiev, I., Křivánek, J., Davidovič, T., and Slusallek, P. (2012). Light transport simulation with vertex connection and merging. *ACM Transactions on Graphics (TOG)*, 31:192:1–192:10.

[Georgiev et al., 2011] Georgiev, I., Křivánek, J., and Slusallek, P. (2011). Bidirectional light transport with vertex merging. In *Siggraph Asia 2011 Sketches*, pages 27:1–27:2, New York, NY, USA. ACM.

[Glassner, 1989] Glassner, A. S. (1989). *An introduction to ray tracing*. Elsevier.

[Gobbetti and Marton, 2005] Gobbetti, E. and Marton, F. (2005). Far voxels: a multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms. *ACM Transactions on Graphics (TOG)*, 24:878–885.

[Goldsmith and Salmon, 1987] Goldsmith, J. and Salmon, J. (1987). Automatic creation of object hierarchies for ray tracing. In *Proceedings Computer Graphics and Applications*, volume 7, pages 14–20. IEEE.

[Greiner, 2004] Greiner, M. E. C. V. G. (2004). Stack implementation on programmable graphics hardware. In *Proceedings Vision, Modeling and Visualization (VMV)*, page 255. IOS Press.

[Gu et al., 2018] Gu, F., Jendersie, J., and Grosch, T. (2018). Fast and dynamic construction of bounding volume hierarchies based on loose octrees. In *Proceedings Vision, Modeling and Visualization (VMV)*. The Eurographics Association.

[Gu et al., 2013] Gu, Y., He, Y., Fatahalian, K., and Blelloch, G. (2013). Efficient bvh construction via approximate agglomerative clustering. In *Proceedings High Performance Graphics (HPG)*, pages 81–88. ACM.

[Gunther et al., 2007] Gunther, J., Popov, S., Seidel, H.-P., and Slusallek, P. (2007). Realtime ray tracing on gpu with bvh-based packet traversal. In *IEEE Symposium on Interactive Ray Tracing (IRT)*, pages 113–118.

[Hachisuka et al., 2008] Hachisuka, T., Ogaki, S., and Jensen, H. W. (2008). Progressive photon mapping. *ACM Transactions on Graphics (TOG)*, 27:130.

[Haines and Wallace, 1994] Haines, E. A. and Wallace, J. R. (1994). Shaft culling for efficient ray-cast radiosity. In *Proceedings Eurographics Workshop on Rendering*, pages 122–138. Springer.

[Hapala et al., 2011a] Hapala, M., Davidovič, T., Wald, I., Havran, V., and Slusallek, P. (2011a). Efficient stack-less bvh traversal for

ray tracing. In *Proceedings Spring Conference on Computer Graphics (SCCG)*, pages 7–12. ACM.

[Hapala and Havran, 2011] Hapala, M. and Havran, V. (2011). Kd-tree traversal algorithms for ray tracing. In *Computer Graphics Forum*, volume 30, pages 199–213. Wiley Online Library.

[Hapala et al., 2011b] Hapala, M., Karlik, O., and Havran, V. (2011b). When it makes sense to use uniform grids for ray tracing. In *Proceedings Winter School of Computer Graphics (WSCG), Communication Papers*, pages 193–200.

[Hasenfratz et al., 2003] Hasenfratz, J.-M., Lapierre, M., Holzschuch, N., and Sillion, F. (2003). A survey of real-time soft shadows algorithms. In *Computer Graphics Forum*, volume 22, pages 753–774. Wiley Online Library.

[Havran, 2000] Havran, V. (2000). *Heuristic ray shooting algorithms*. PhD thesis, Ph.D. Thesis, Czech Technical University in Prague.

[Havran et al., 2006] Havran, V., Herzog, R., and Seidel, H.-P. (2006). On the fast construction of spatial hierarchies for ray tracing. In *IEEE Symposium on Interactive Ray Tracing (IRT)*, pages 71–80.

[Heckbert and Hanrahan, 1984] Heckbert, P. S. and Hanrahan, P. (1984). Beam tracing polygonal objects. *ACM Siggraph Computer Graphics*, 18:119–127.

[Hendrich et al., 2017] Hendrich, J., Meister, D., and Bittner, J. (2017). Parallel bvh construction using progressive hierarchical refinement. In *Computer Graphics Forum*, volume 36, pages 487–494. Wiley Online Library.

[Horn et al., 2007] Horn, D. R., Sugerman, J., Houston, M., and Hanrahan, P. (2007). Interactive kd tree gpu raytracing. In *Symposium on Interactive 3D Graphics and Games (I3D)*, pages 167–174. ACM.

[Jensen, 1996] Jensen, H. W. (1996). Global illumination using photon maps. In *Rendering Techniques' 96*, pages 21–30. Springer.

[Jevans and Wyvill, 1989] Jevans, D. and Wyvill, B. (1989). Adaptive voxel subdivision for ray tracing. In *Proceedings Graphics Interface*, pages 164–172. Canadian Man-Computer Communications Society.

[Kajiya, 1986] Kajiya, J. T. (1986). The rendering equation. *ACM Siggraph Computer Graphics*, 20:143–150.

[Kalojanov et al., 2011] Kalojanov, J., Billeter, M., and Slusallek, P. (2011). Two-level grids for ray tracing on gpus. In *Computer Graphics Forum*, volume 30, pages 307–314. Wiley Online Library.

[Kalojanov and Slusallek, 2009] Kalojanov, J. and Slusallek, P. (2009). A parallel algorithm for construction of uniform grids. In *Proceedings High Performance Graphics (HPG)*, pages 23–28. ACM.

[Kammaje and Mora, 2007] Kammaje, R. P. and Mora, B. (2007). A study of restricted bsp trees for ray tracing. In *IEEE Symposium on Interactive Ray Tracing (IRT)*, volume 0, pages 55–62, Los Alamitos, CA, USA.

[Kämpe et al., 2013] Kämpe, V., Sintorn, E., and Assarsson, U. (2013). High resolution sparse voxel dags. *ACM Transactions on Graphics (TOG)*, 32:101.

[Kaplan, 1985] Kaplan, M. R. (1985). Space-tracing: A constant time ray-tracer. *ACM Siggraph Computer Graphics*.

[Karras, 2012] Karras, T. (2012). Maximizing parallelism in the construction of bvhs, octrees, and k-d trees. In *Proceedings High Performance Graphics (HPG)*, pages 33–37. Eurographics Association.

[Karras and Aila, 2013] Karras, T. and Aila, T. (2013). Fast parallel construction of high-quality bounding volume hierarchies. In *Proceedings High Performance Graphics (HPG)*, pages 89–99. ACM.

[Kay and Kajiya, 1986] Kay, T. L. and Kajiya, J. T. (1986). Ray tracing complex scenes. *ACM Siggraph Computer Graphics*, 20:269–278.

[Kensler, 2008] Kensler, A. (2008). Tree rotations for improving bounding volume hierarchies. In *IEEE Symposium on Interactive Ray Tracing (IRT)*, pages 73–76.

[Keul et al., 2017] Keul, K., Klee, N., and Müller, S. (2017). Soft shadow computation using precomputed line space visibility information. *Journal of International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG)*, 25:97–106.

[Keul et al., 2018] Keul, K., Koß, T., and Müller, S. (2018). Fast indirect lighting approximations using the representative candidate line space. *Journal of International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG)*, 26:11–20.

[Keul et al., 2019] Keul, K., Koß, T., Schröder, F. L., and Müller, S. (2019). Combining two-level data structures and line space precomputations to accelerate indirect illumination. In *Computer Vision, Imaging and Computer Graphics Theory and Applications*.

[Keul et al., 2016] Keul, K., Müller, S., and Lemke, P. (2016). Accelerating spatial data structures in ray tracing through precomputed line space visibility. In *Proceedings International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG)*, volume 24, pages 17–25.

[Kwon et al., 1998] Kwon, B., Kim, D. S., Chwa, K.-Y., and Shin, S. Y. (1998). Memory-efficient ray classification for visibility operations. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 4:193–201.

[Lafortune and Willems, 1993] Lafortune, E. P. and Willems, Y. D. (1993). Bi-directional path tracing. In *Proceedings 3. International Conference on Computational Graphics and Visualization Techniques*, pages 145–153, Alvor, Portugal.

[Laine et al., 2005] Laine, S., Aila, T., Assarsson, U., Lehtinen, J., and Akenine-Möller, T. (2005). Soft shadow volumes for ray tracing. *ACM Transactions on Graphics (TOG)*, 24:1156–1165.

[Laine and Karras, 2011] Laine, S. and Karras, T. (2011). Efficient sparse voxel octrees. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 17:1048–1059.

[Laine et al., 2009] Laine, S., Siltanen, S., Lokki, T., and Savioja, L. (2009). Accelerated beam tracing algorithm. *Applied Acoustics*, 70:172–181.

[Lauterbach et al., 2009] Lauterbach, C., Garland, M., Sengupta, S., Luebke, D., and Manocha, D. (2009). Fast bvh construction on gpus. In *Computer Graphics Forum*, volume 28, pages 375–384. Wiley Online Library.

[Lauterbach et al., 2006] Lauterbach, C., Yoon, S.-E., Manocha, D., and Tuft, D. (2006). Rt-deform: Interactive ray tracing of dynamic

scenes using bvhs. In *IEEE Symposium on Interactive Ray Tracing (IRT)*, pages 39–46.

[Leyvand et al., 2003] Leyvand, T., Sorkine, O., and Cohen-Or, D. (2003). *Ray space factorization for from-region visibility*, volume 22. ACM.

[Li and Liu, 2018] Li, H. and Liu, W. (2018). Accurate shadow generation analysis in computer graphics. In *IEEE 20th High Performance Computing and Communications*, pages 1116–1120.

[Meister and Bittner, 2016] Meister, D. and Bittner, J. (2016). Parallel bvh construction using k-means clustering. *The Visual Computer*, 32:977–987.

[Meister and Bittner, 2017] Meister, D. and Bittner, J. (2017). Parallel locally-ordered clustering for bounding volume hierarchy construction. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*.

[Meister and Bittner, 2018] Meister, D. and Bittner, J. (2018). Parallel reinsertion for bounding volume hierarchy optimization. In *Computer Graphics Forum*, volume 37, pages 463–473. Wiley Online Library.

[Méndez-Feliu and Sbert, 2009] Méndez-Feliu, À. and Sbert, M. (2009). From obscurances to ambient occlusion: A survey. *The Visual Computer*, 25:181–196.

[Mortensen et al., 2007] Mortensen, J., Khanna, P., Yu, I., and Slater, M. (2007). A visibility field for ray tracing. In *Computer Graphics, Imaging and Visualisation (CGIV)*, pages 54–61. IEEE.

[Pantaleoni and Luebke, 2010] Pantaleoni, J. and Luebke, D. (2010). Hlbvh: hierarchical lbvh construction for real-time ray tracing of dynamic geometry. In *Proceedings High Performance Graphics (HPG)*, pages 87–95. Eurographics Association.

[Parker et al., 2010] Parker, S. G., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robison, A., et al. (2010). Optix: a general purpose ray tracing engine. *ACM Transactions on Graphics (TOG)*, 29:66.

[Pérard-Gayot et al., 2017] Pérard-Gayot, A., Kalojanov, J., and Slusallek, P. (2017). Gpu ray tracing using irregular grids. In

Computer Graphics Forum, volume 36, pages 477–486. Wiley Online Library.

[Pharr et al., 2016] Pharr, M., Jakob, W., and Humphreys, G. (2016). *Physically based rendering: From theory to implementation*. Morgan Kaufmann.

[Popov et al., 2009] Popov, S., Georgiev, I., Dimov, R., and Slusallek, P. (2009). Object partitioning considered harmful: Space subdivision for bvhs. In *Proceedings High Performance Graphics (HPG)*, pages 15–22, New York, NY, USA. ACM.

[Popov et al., 2006] Popov, S., Gunther, J., Seidel, H.-P., and Slusallek, P. (2006). Experiences with streaming construction of sah kd-trees. In *IEEE Symposium on Interactive Ray Tracing (IRT)*, pages 89–94.

[Popov et al., 2007] Popov, S., Günther, J., Seidel, H.-P., and Slusallek, P. (2007). Stackless kd-tree traversal for high performance gpu ray tracing. In *Computer Graphics Forum*, volume 26, pages 415–424. Wiley Online Library.

[Reeves et al., 1987] Reeves, W. T., Salesin, D. H., and Cook, R. L. (1987). Rendering antialiased shadows with depth maps. In *ACM Siggraph Computer Graphics*, volume 21, pages 283–291.

[Ren et al., 2005] Ren, Z., Hua, W., Chen, L., and Bao, H. (2005). Intersection fields for interactive global illumination. *The Visual Computer*, 21:569–578.

[Reshetov et al., 2005] Reshetov, A., Soupikov, A., and Hurley, J. (2005). Multi-level ray tracing algorithm. *ACM Transactions on Graphics (TOG)*, 24:1176–1185.

[Revelles et al., 2000] Revelles, J., Ureña, C., and Lastra, M. (2000). An efficient parametric algorithm for octree traversal. *Journal of Winter School of Computer Graphics (WSCG)*, 8:212–219.

[Ritschel et al., 2012] Ritschel, T., Dachsbacher, C., Grosch, T., and Kautz, J. (2012). The state of the art in interactive global illumination. In *Computer Graphics Forum*, volume 31, pages 160–188. Wiley Online Library.

[Roccia et al., 2012] Roccia, J.-P., Coustet, C., and Paulin, M. (2012). Hybrid cpu/gpu kd-tree construction for versatile ray tracing. In *Eurographics*.

[Samet, 1989] Samet, H. (1989). Implementing ray tracing with octrees and neighbor finding. *Computers & Graphics*, 13:445–460.

[Schmittler et al., 2004] Schmittler, J., Woop, S., Wagner, D., Paul, W. J., and Slusallek, P. (2004). Realtime ray tracing of dynamic scenes on an fpga chip. In *Proceedings ACM Siggraph/Eurographics Graphics hardware*, pages 95–106. ACM.

[Schwarz and Seidel, 2010] Schwarz, M. and Seidel, H.-P. (2010). Fast parallel surface and solid voxelization on gpus. *ACM Transactions on Graphics (TOG)*, 29:179.

[Shanmugam and Arikan, 2007] Shanmugam, P. and Arikan, O. (2007). Hardware accelerated ambient occlusion techniques on gpus. In *Symposium on Interactive 3D Graphics and Games (I3D)*, pages 73–80. ACM.

[Sharpe et al., 2003] Sharpe, A., Hampton, M., Nirenstein, S., Gain, J., and Blake, E. (2003). Accelerating ray shooting through aggressive 5d visibility preprocessing. In *Proceedings Computer graphics, virtual Reality, visualisation and interaction in Africa*, pages 95–100. ACM.

[Shevtsov et al., 2007] Shevtsov, M., Soupikov, A., and Kapustin, A. (2007). Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. In *Computer Graphics Forum*, volume 26, pages 395–404. Wiley Online Library.

[Sintorn et al., 2014] Sintorn, E., Kämpe, V., Olsson, O., and Assarsson, U. (2014). Compact precomputed voxelized shadows. *ACM Transactions on Graphics (TOG)*, 33:150.

[Smits, 1998] Smits, B. (1998). Efficiency issues for ray tracing. *Journal of Graphics Tools*, 3:1–14.

[Stich et al., 2009] Stich, M., Friedrich, H., and Dietrich, A. (2009). Spatial splits in bounding volume hierarchies. In *Proceedings High Performance Graphics (HPG)*, pages 7–13. ACM.

[Sung and Shirley, 1992] Sung, K. and Shirley, P. (1992). Ray tracing with the bsp tree. In *Graphics Gems III*, pages 271–274. Academic Press.

[Thiago Ize, 2008] Thiago Ize, Ingo Wald, S. G. P. (2008). Ray tracing with the bsp tree. In *IEEE Symposium on Interactive Ray Tracing (IRT)*, pages 159–166.

[Thiedemann et al., 2011] Thiedemann, S., Henrich, N., Grosch, T., and Müller, S. (2011). Voxel-based global illumination. In *Symposium on Interactive 3D Graphics and Games (I3D)*, pages 103–110, New York, NY, USA. ACM.

[Tsakok, 2009] Tsakok, J. A. (2009). Faster incoherent rays: Multi-bvh ray stream tracing. In *Proceedings High Performance Graphics (HPG)*, pages 151–158. ACM.

[Veach, 1998] Veach, E. (1998). *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, Stanford University, Stanford, CA, USA.

[Veach and Guibas, 1995a] Veach, E. and Guibas, L. (1995a). Bidirectional estimators for light transport. In *Photorealistic Rendering Techniques*, pages 145–167. Springer.

[Veach and Guibas, 1995b] Veach, E. and Guibas, L. J. (1995b). Optimally combining sampling techniques for monte carlo rendering. In *Proceedings 22. Annual Conference on Computer Graphics and Interactive Techniques*, pages 419–428, New York, NY, USA. ACM.

[Vinkler et al., 2017] Vinkler, M., Bittner, J., and Havran, V. (2017). Extended morton codes for high performance bounding volume hierarchy construction. In *Proceedings High Performance Graphics (HPG)*, page 9. ACM.

[Vinkler et al., 2014] Vinkler, M., Havran, V., and Bittner, J. (2014). Bounding volume hierarchies versus kd-trees on contemporary many-core architectures. In *Proceedings Spring Conference on Computer Graphics (SCCG)*, pages 29–36, New York, NY, USA. ACM.

[Vinkler et al., 2016] Vinkler, M., Havran, V., and Bittner, J. (2016). Performance comparison of bounding volume hierarchies and kd-trees for gpu ray tracing. In *Computer Graphics Forum*, volume 35, pages 68–79. Wiley Online Library.

[Wächter and Keller, 2006] Wächter, C. and Keller, A. (2006). Instant ray tracing: The bounding interval hierarchy. *Rendering Techniques*, 2006:139–149.

[Wald, 2005] Wald, I. (2005). Realtime ray tracing and interactive global illumination. *Ausgezeichnete Informatikdissertationen 2004*.

[Wald, 2007] Wald, I. (2007). On fast construction of sah-based bounding volume hierarchies. In *IEEE Symposium on Interactive Ray Tracing (IRT)*, pages 33–40.

[Wald, 2012] Wald, I. (2012). Fast construction of sah bvhs on the intel many integrated core (mic) architecture. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 18:47–57.

[Wald et al., 2008] Wald, I., Benthin, C., and Boulos, S. (2008). Getting rid of packets-efficient simd single-ray traversal using multi-branching bvhs. In *IEEE Symposium on Interactive Ray Tracing (IRT)*, pages 49–57.

[Wald et al., 2003a] Wald, I., Benthin, C., and Slusallek, P. (2003a). Distributed interactive ray tracing of dynamic scenes. In *IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, page 11.

[Wald et al., 2007] Wald, I., Boulos, S., and Shirley, P. (2007). Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics (TOG)*, 26:6.

[Wald and Havran, 2006] Wald, I. and Havran, V. (2006). On building fast kd-trees for ray tracing, and on doing that in o (n log n). In *IEEE Symposium on Interactive Ray Tracing (IRT)*, pages 61–69.

[Wald et al., 2006] Wald, I., Ize, T., Kensler, A., Knoll, A., and Parker, S. G. (2006). Ray tracing animated scenes using coherent grid traversal. *ACM Transactions on Graphics (TOG)*, 25:485–493.

[Wald et al., 2009] Wald, I., Mark, W. R., Günther, J., Boulos, S., Ize, T., Hunt, W., Parker, S. G., and Shirley, P. (2009). State of the art in ray tracing animated scenes. In *Computer Graphics Forum*, volume 28, pages 1691–1722. Wiley Online Library.

[Wald et al., 2003b] Wald, I., Purcell, T. J., Schmittler, J., Benthin, C., and Slusallek, P. (2003b). Realtime ray tracing and its use for interactive global illumination. *Eurographics State of the Art Reports*, 1:5.

[Wald et al., 2001] Wald, I., Slusallek, P., Benthin, C., and Wagner, M. (2001). Interactive rendering with coherent ray tracing. In *Computer Graphics Forum*, volume 20, pages 153–165. Wiley Online Library.

[Wald et al., 2014] Wald, I., Woop, S., Benthin, C., Johnson, G. S., and Ernst, M. (2014). Embree: a kernel framework for efficient cpu ray tracing. *ACM Transactions on Graphics (TOG)*, 33:143.

[Walter et al., 2008] Walter, B., Bala, K., Kulkarni, M., and Pingali, K. (2008). Fast agglomerative clustering for rendering. In *IEEE Symposium on Interactive Ray Tracing (IRT)*, pages 81–86.

[Wang et al., 2016] Wang, Y., Guo, P., and Duan, F. (2016). A fast ray tracing algorithm based on a hybrid structure. *Multimedia Tools and Applications*, 75:1883–1898.

[Williams, 1978] Williams, L. (1978). Casting curved shadows on curved surfaces. In *Proceedings Computer Graphics and Interactive Techniques*, volume 12, pages 270–274, New York, NY, USA. ACM.

[Wodniok and Goesele, 2017] Wodniok, D. and Goesele, M. (2017). Construction of bounding volume hierarchies with sah cost approximation on temporary subtrees. *Computers & Graphics*, 62:41–52.

[Woop et al., 2005] Woop, S., Schmittler, J., and Slusallek, P. (2005). Rpu: a programmable ray processing unit for realtime ray tracing. *ACM Transactions on Graphics (TOG)*, 24:434–444.

[Yang et al., 2010] Yang, B., Dong, Z., Feng, J., Seidel, H.-P., and Kautz, J. (2010). Variance soft shadow mapping. In *Computer Graphics Forum*, volume 29, pages 2127–2134. Wiley Online Library.

[Yin and Li, 2014] Yin, M. and Li, S. (2014). Fast bvh construction and refit for ray tracing of dynamic scenes. *Multimedia Tools and Applications*, 72:1823–1839.

[Ylitie et al., 2017] Ylitie, H., Karras, T., and Laine, S. (2017). Efficient incoherent ray traversal on gpus through compressed wide bvhs. In *Proceedings High Performance Graphics (HPG)*, page 4. ACM.

[Yu et al., 2009] Yu, I., Cox, A., Kim, M. H., Ritschel, T., Grosch, T., Dachsbacher, C., and Kautz, J. (2009). Perceptual influence of approximate visibility in indirect illumination. *ACM Transactions on Applied Perception (TAP)*, 6:24.

[Zachmann, 2002] Zachmann, G. (2002). Minimal hierarchical collision detection. In *Symposium on Virtual Reality Software and Technology (VRST)*, pages 121–128, Hong Kong, China. ACM.

[Zlatuška and Havran, 2010] Zlatuška, M. and Havran, V. (2010). Ray tracing on a gpu with cuda–comparative study of three algorithms. In *Proceedings Winter School of Computer Graphics (WSCG)*.

DATEN

E-MAIL
keul@uni-koblenz.de

GEBURTSORT
Koblenz

GEBURTSTAG
19.12.1990

CURRICULUM VITAE

KEVIN KEUL

WERDEGANG

04/2014 - 09/2019 **Wissenschaftlicher Mitarbeiter**
Universität Koblenz-Landau - Arbeitsgruppe für Computergrafik

- Forschung und Entwicklung im Bereich von GPU-Datenstrukturen zur Raytracing-Beschleunigung
- Tutor für diverse Lehrveranstaltungen in den Bereichen Computergrafik, Echtzeitrendering und Fotorealistisches Rendering
- Betreuung diverser studentischer Seminare, Abschlussarbeiten und Praktika
- Entwicklung eines Rendering-Frameworks für Lehre und Abschlussarbeiten
- Wiederholte Organisation eines Messestandes auf der Computer- und Videospielmesse Gamescom

10/2012 - 03/2014 **Wissenschaftliche Hilfskraft**
Universität Koblenz-Landau - Arbeitsgruppe für Computergrafik

- Entwicklung von Codebeispielen und Unterlagen für die Lehre
- Durchführung einer Lehrveranstaltung
- Korrektur von Übungsabgaben

10/2012 - 03/2013 **Wissenschaftliche Hilfskraft**
Universität Koblenz-Landau - Arbeitsgruppe für Softwaretechnik

- Korrektur von Übungsabgaben
- Betreuung eines Java-Einstiegskurses

VORTRÄGE UND VERÖFFENTLICHUNGEN

2019 **K.Keul**, T.Koß, F.L.Schröder and S.Müller: *Combining Two-Level Data Structures and Line Space Precomputations to accelerate Indirect Illumination* (Grapp, 2019)

2018 **K.Keul**, T.Koß, S.Müller: *Fast Indirect Lighting Approximations using the Representative Candidate Line Space* (WSCG, 2018)

2017 **K.Keul**, N.Klee, S.Müller: *Soft Shadow Computation using Precomputed Line Space Visibility Information* (WSCG, 2017)

2016 **K.Keul**, P.Lemke, S.Müller: *Accelerating Spatial Data Structures in Ray Tracing through Precomputed Line Space Visibility* (WSCG, 2016)

SCHULISCHE UND AKADEMISCHE AUSBILDUNG

Seit 04/2014 **Dr. rer. nat. Informatik**
Universität Koblenz-Landau, Voraussichtliche Abschlussarbeit: *The Line Space - a Directional Data Structure for Ray Tracing Acceleration*

10/2012 - 03/2014 **Master of Science Informatik** (Vertiefung Software-Engineering)
Universität Koblenz-Landau (Abschlussnote 1,5), Abschlussarbeit: *GPGPU im Kontext der molekularen Dynamik*

04/2010 - 09/2012 **Bachelor of Science Informatik** (Nebenfach Mathematik)
Universität Koblenz-Landau (Abschlussnote 1,7), Abschlussarbeit: *Weiterentwicklung einer eigenen 3D-Engine und Integration moderner Renderingverfahren*

2010 **Abitur** - Goethe-Gymnasium Bad Ems (Abschlussnote 1,7)