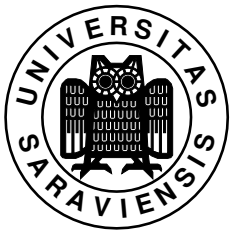
# Realtime Ray Tracing

# on current CPU Architectures

**Carsten Benthin**

**Computer Graphics Group**
**Saarland University**
**66123 Saarbrücken, Germany**

Dissertation zur Erlangung des Grades
*Doktor der Ingenieurwissenschaften (Dr.-Ing.)*
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes



**Betreuender Hochschullehrer / Supervisor:**
Prof. Dr.-Ing. Philipp Slusallek
Universität des Saarlandes, Saarbrücken, Germany

**Gutachter / Reviewers:**
Prof. Dr.-Ing. Philipp Slusallek
Universität des Saarlandes, Saarbrücken, Germany

Prof. Dr. rer. nat. Hans-Peter Seidel
MPI Informatik, Saarbrücken, Germany

Research Assistant Professor Steven G. Parker
University of Utah, Salt Lake City, UT, USA

**Dekan / Dean:**
Prof. Dr. rer. nat. Jörg Eschmeier
Universität des Saarlandes, Saarbrücken, Germany

**Eingereicht am / Thesis submitted:**
30. Januar 2006 / Jan 30st, 2006

Carsten Benthin
Lehrstuhl für Computergraphik, Geb. 36.1/E14
Im Stadtwald, 66123 Saarbrücken
Germany
benthin@graphics.cs.uni-sb.de

# Abstract

In computer graphics, ray tracing has become a powerful tool for generating realistically looking images. Even though ray tracing offers high flexibility, a logarithmic scalability in scene complexity, and is known to be efficiently parallelizable, its demand for compute power has in the past lead to its limitation to high-quality off-line rendering.

This thesis focuses on the question of how realtime ray tracing can be realized on current processor architectures. To this end, it provides a detailed analysis of the weaknesses and strengths of current processor architectures, for the purpose of allowing for highly optimized implementation. The combination of processor-specific optimizations with algorithms that exploit the coherence of ray tracing, makes it possible to achieve realtime performance on a single CPU.

Besides the optimization of the ray tracing algorithm itself, this thesis focuses on the efficient building of spatial index structures. By building these structures from scratch for every frame, interactive ray tracing of fully dynamic scenes becomes possible. Moreover, a parallelization framework for ray tracing is discussed that efficiently exploits the compute power of a cluster of commodity PCs. Finally, a global illumination algorithm is proposed that efficiently combines optimized ray tracing and the parallelization framework. The combination makes it possible to compute complete global illumination at interactive frame rates.

# Kurzfassung

In der Computer-Graphik hat sich Ray-Tracing längst als wichtiges Werkzeug zur realistischen Bildsynthese etabliert. Entscheidend dazu beigetragen haben dessen Flexibilität und logarithmische Skalierung in der Szenengröße, sowie die effiziente Parallelisierbarkeit. Aufgrund der hohen Anforderung an Rechenkapazität war die Verwendung bisher auf den qualitativ hochwertigen, aber nicht interaktiven Bereich der realistischen Bildsynthese beschränkt.

Diese Dissertation beschäftigt sich mit der Frage, wie die Geschwindigkeit von Ray-Tracing auf heutigen Prozessorarchitekturen derart gesteigert werden kann, dass es die Bildsynthese in Echtzeit ermöglicht. Dazu präsentiert die vorliegende Arbeit eine genaue Analyse der Stärken und Schwächen der heutigen Prozessorarchitekturen, um die benötigten Algorithmen entsprechend zu optimieren. Darauf aufbauend werden Algorithmen vorgestellt, die es im besonderen Maße erlauben, Kohärenz innerhalb des Ray-Tracing Verfahrens effizient auszunutzen. Diese Kombination von kohärenz-ausnutzenden Algorithmen mit einer prozessoroptimierten Implementierung ermöglicht sogar die interaktive Bildsynthese bei Ausnutzung der Rechenkapazität eines einzelnen Prozessors.

Darüber hinaus präsentiert die vorliegende Arbeit einen neuen Algorithmus, der die Zeit für den Aufbau der für das Ray-Tracing benötigten räumlichen Beschleunigungsdatenstrukturen erheblich verkürzt. Der beschleunigte Aufbau erlaubt sogar das interaktive Ray-Tracing von vollständig dynamischen Szenen. Daneben wird ein Parallelisierungsystem für Ray-Tracing vorgestellt, welches die Rechenkapazität eines Netzwerkes von Standardrechnern effizient kombiniert, um sogar Bildsynthese in Echtzeit zu erreichen. Abschließend wird ein Verfahren zur physikalisch korrekten Beleuchtungssimulation beschrieben, welches bereits vorgestellte Techniken wie optimiertes Ray-Tracing und effiziente Parallelisierung verbindet. Diese Kombination ermöglicht es letztendlich die physikalisch korrekte Beleuchtung mehrmals pro Sekunde komplett neu zu berechnen.

# Zusammenfassung

In der Computer-Graphik hat sich Ray-Tracing längst als wichtiges Werkzeug zur realistischen Bildsynthese etabliert. Entscheidend dazu beigetragen haben die Flexibilität und logarithmische Skalierung in der Szenengröße sowie die effiziente Parallelisierbarkeit des Ray-Tracing Verfahrens an sich. Aufgrund der langen Laufzeit und der hohen Anfordung an Rechenkapazität war die Verwendung von Ray-Tracing bisher auf den qualitativ hochwertigen, aber nicht interaktiven Bereich der realistischen Bildsynthese beschränkt.

Diese Dissertation beschäftigt sich mit der Frage, wie die Geschwindigkeit von Ray-Tracing derart gesteigert werden kann, dass Ray-Tracing auch für die interaktive Bildsynthese bzw. die Bildsynthese in Echtzeit geeignet ist. Als ersten Schritt dazu präsentiert die vorliegende Arbeit eine genaue Analyse der heutigen Prozessorarchitekturen, die die zugrundeliegende Hardware-Plattform bilden. Dabei werden deren Stärken und Schwächen detailliert aufgezeigt und daraus abgeleitet Implementierungs- und Optimierungsrichtlinien vorgestellt. Diese Richtlinien erlauben es, ineffizienten Code bei der Implementierung des Ray-Tracing Verfahrens zu vermeiden.

Einer der Hauptschwerpunkte der vorliegenden Disseration liegt auf der Entwicklung von Algorithmen, die es erlauben Kohärenz innerhalb des Ray-Tracing Verfahrens effizient auszunutzen. Gerade die Anwendung von Operationen auf kohärente Strahlbündel anstatt auf einzelne Strahlen ermöglicht eine erhebliche Steigerung der Geschwindigkeit von Ray-Tracing. Dies wird detailliert an den zwei fundamentalen Algorithmen des Ray-Tracing Verfahrens, der Traversierung von Strahlen durch eine räumliche Beschleunigungsdatenstruktur und dem Schnittpunkttest zwischen Strahl und geometrischem Primitive aufgezeigt. Beim Schnittpunkttest wird neben der Unterstützung für Dreiecke ein besonderes Augenmerk auf die effiziente Unterstützung von Freiformflächen gelegt. Im Gegensatz zur Beschreibung einer Szene mittels Dreiecken erlaubt die Beschreibung mittels Freiformflächen eine viel kompaktere und genauere Repräsentation. Allerdings gestaltet sich der benötigte Schnittpunkttest ungleich aufwendiger. Diese Arbeit stellt dazu verschiedene Algorithmen vor, die je nach Anwendungsgebiet und Genauigkeitsanforderungen unterschiedlich eingesetzt werden können. Die Kombination kohärenz-ausnutzender Algorithmen zur Traversierung und Schnittpunktberechnung mit einer effizienten und prozessornahen Implementierung ermöglicht sogar die interaktive Bildsynthese bei Ausnutzung der Rechenkapazität eines einzelnen Prozessors.

Neben der Optimierung des Ray-Tracing Verfahrens an sich, stellt diese Dissertation ein Algorithmus vor, um die für Ray-Tracing benötigten räumlichen Beschleunigungsdatenstrukturen effizient aufzubauen. Dabei wird auf

dieselben Optimierungsstrategien zurückgegriffen, die bereits bei der Beschleunigung des Ray-Tracing Verfahrens zum Tragen kommen. Der optimierte Aufbaualgorithmus erlaubt sogar das interaktive Ray-Tracing von vollständig dynamischen Szenen, indem die Beschleunigungsdatenstrukturen mehrmals pro Sekunde komplett neu aufgebaut werden.

Weiterhin wird ein System zur Parallelisierung des Ray-Tracing Verfahrens präsentiert, das die Rechenkapazität eines Netzwerkes von Standardrechnern effizient kombiniert. Dabei wird das System mit dem Ziel entwickelt, die Nachteile einer solchen verteilten Architektur, wie beispielsweise getrennte Hauptspeicher und langsame Verbindungsbandbreiten, effizient zu kompensieren. So gelingt es, die Geschwindigkeit von Ray-Tracing linear mit der Anzahl der verbundenen Rechner zu steigern, wodurch Ray-Tracing sogar in Echtzeit ermöglicht wird.

Im letzten Teil der Dissertation wird ein Verfahren zur physikalisch korrekten Beleuchtungssimulation vorgestellt, welches die bereits vorgestellte Techniken, hoch optimierter Ray-Tracing Kern und Rahmenwerk zur Parallelisierung, effektiv verbindet. Die Kombination dieser Techniken mit einem auf Kohärenzausnutzung ausgelegtem Algorithmus zur Beleuchtungssimulation ermöglicht es letztendlich, die physikalisch korrekte Beleuchtung mehrmals pro Sekunde komplett neu zu berechnen.

# Acknowledgements

First of all, I would like to thank Prof. Dr. Philipp Slusallek for supervising this thesis. The open and encouraging atmosphere in his group and in particular his continuous support were invaluable for the success of my thesis.

Second, I have to thank Dr. Ingo Wald, who taught me basically everything I know about ray tracing. He has been an invaluable help in many projects related to this thesis. Without his encouraging support and a countless number of discussions through the years I would not have been able to complete my PhD thesis.

I would also like to thank my reviewers, Hans-Peter Seidel and Steven Parker, for kindly accepting the responsibility of reviewing my thesis.

Similarly, I have to thank (in random order) Andreas Dietrich, Heiko Friedrich, Johannes Günther, Jörg Schmittler, and Georg Demme and his administration group for many fruitful discussions, ideas and help in many projects. Special thanks goes to Michael Scherbaum for reducing my workload during writing this thesis. Furthermore, I would like to thank the current and former members and students of the computer graphics group.

I would also like to thank James T. Hurley for giving me the opportunity to do an internship at Intel Corp. Many thanks are due to Gordon Stoll and Alexander Reshetov for many helpful discussions, for introducing me to many new ideas, and for simply making the stay a great experience.

Special thanks goes to my sister Nicole Benthin, who helped me writing this thesis in 'readable' English.

Finally, and most importantly, I would like to thank my familiy, and in particular my wife Andrea for their great patience, their encouraging support and for bearing so many stressful times. Without their help this thesis would never have been possible.

# Contents

# List of Figures

# List of Tables

# Chapter 1
# Introduction

In the context of computer graphics, the term *rendering* refers to the process of generating a two-dimensional image from a three-dimensional virtual scene. Rendering forms the basis for many fields of today's computer graphics, e.g. computer games, visualization, and graphical effects used for movie productions. Based on the algorithm used for the rendering process, two major rendering categories can be classified: rasterization-based rendering and ray tracing-based rendering.

For decades, ray tracing has been used exclusively for high-quality rendering, where it has been known for its long rendering times. Therefore, ray tracing's sole application has been off-line rendering. On the other hand, the field of interactive rendering has been dominated by rasterization-based hardware rendering.

Beyond any doubt, the key factor for the non-existence of ray tracing in terms of interactive rendering has been its poor performance. Researchers have long argued that thanks to its logarithmic behavior in scene complexity, ray tracing could eventually become faster than rasterization-based rendering; nevertheless its performance has been far from challenging. However, if enough parallel compute power was available, even the performance of ray tracing has been able to reach interactivity [Keates95, Muuss95a, Muuss95b, Parker99b]. Unfortunately, a large scale supercomputer was required to provide enough compute power.

In recent years, researchers have once again focused on the performance of ray tracing [Wald01c, Wald03e, Wald04, Reshetov05], in particular with a focus on off-the-shelf hardware. They have concentrated on an efficient and optimized implementation of the ray tracing algorithm and its data structures with respect to the advantages and disadvantages of current hardware architectures. The efficient combination of algorithmic and hardware-specific

optimizations has allowed ray tracing's performance to be lifted to an interactive level while running on off-the-shelf hardware [Wald04].

Note that implementations of ray tracing exist for other architectures such as GPUs [Purcell02, Foley05], or even custom hardware [Schmittler02, Schmittler04, Woop05]. However, they are still too slow (GPUs) or not widely available (custom hardware), so the current processor architectures are considered as the most suitable hardware platform.

The main difficulty when using processor-specific optimized algorithms in order to achieve high performance ray tracing is that they are not easy to explain in general, especially not from a high-level view.

This thesis presents the latest approaches and algorithms for realtime ray tracing on current processor architectures. The intention when writing this thesis was to avoid a strictly high-level view but to give exact and detailed information about low-level implementation issues and optimizations. Where possible, example code is provided for each illustrated algorithm.

The main contributions of this thesis to the field of ray tracing are

- SIMD-optimized algorithms for the traversal of coherent rays,
- a detailed analysis and SIMD-optimized implementation of intersection algorithms for rays with different geometric primitives such as triangles and bicubic Bézier patches,
- an algorithm for handling fully dynamic scenes by fast building of spatial index structures,
- a parallelization framework for ray tracing that achieves linear scalability in the number of connected computing nodes,
- and a ray tracing-based system that allows for interactively computing global illumination.

All algorithms illustrated in this thesis are an integral part of the OpenRT ray tracing library [Wald02a, Dietrich03, Wald04].

Even though some of the techniques discussed in this thesis have already been sketched in [Wald04], they will be discussed here in more detail, while, in particular, focusing on low-level implementation aspects.

The code examples provided in this thesis have already been optimized for high performance, but there is still much room for further optimization. Even though optimizing might take time and "cost nerves", it can sometimes be the key factor in order to lift algorithms to a new performance level.

## 1.1 Outline of this thesis

The thesis starts with a brief introduction to ray tracing and its use for rendering in Chapter 2. The chapter also provides a high-level performance analysis of the ray tracing algorithm and introduces the benefits of coherence and CPU architecture-specific optimizations.

Chapter 3 discusses current CPU architectures in detail. As these architectures build the underlying hardware platform for the implementation of all algorithms proposed in this thesis, special emphasis is put on performance issues and useful coding guidelines. The chapter introduces SIMD instructions, which will be an essential tool for exploiting the full compute power of current CPU architectures.

Increasing ray tracing performance in particular requires an efficient traversal through a spatial index structure. Therefore, Chapter 4 proposes highly optimized algorithms for efficient traversal of coherent sets of rays. In combination with optimized triangle-intersection algorithms that have been modified to efficiently support coherent sets of rays, as illustrated in Chapter 5, interactive ray tracing performance for ray tracing of triangular scenes will be achieved on a single processor.

Chapter 6 shows that interactive ray tracing is not limited to triangular scenes and presents various highly efficient intersection algorithms for bicubic Bézier patches in detail. These algorithms will be discussed in depth showing their advantages and disadvantages in terms of performance and accuracy.

Chapter 7 demonstrates that even the construction algorithm for spatial index structures can be efficiently optimized. A highly optimized implementation in particular allows for handling fully dynamic scenes by reconstructing the corresponding kd-tree from scratch for every frame.

Compensating the need of ray tracing for compute power means combining the compute power of multiple processors. Chapter 8 thus proposes a parallelization framework that effectively distributes the rendering work to a cluster of off-the-shelf PCs. As this framework has been designed for handling high-latency interconnections, linear scalability in the number of connected PCs will be achieved, increasing the performance of ray-tracing to a realtime level.

If the techniques for achieving realtime ray tracing, as discussed in the previous chapters, are combined with a global illumination algorithm that has been exclusively modified to exploit these techniques, it becomes possible to achieve interactive global illumination. Chapter 9 illustrates this modified global illumination algorithm and the surrounding framework.

Finally, this thesis ends with a short summary, and an outlook on the future of realtime ray tracing on future processor architectures.

# Chapter 2
# Introduction to Ray Tracing

This chapter starts with a brief review of the basic ray tracing algorithms (see Section 2.1) in order to provide a quick introduction to the field of high performance ray tracing. For a more detailed introduction to the field of ray tracing, please refer to one of the classical ray tracing books [Glassner89, Glassner95, Shirley03, Pharr04].

After illustrating how ray tracing is efficiently applied as a rendering algorithm (see Section 2.2), general performance and optimization techniques are discussed (see Section 2.3). These consolidated findings on ray tracing performance allows better access to the role of coherence (see Section 2.4). Besides hardware-specific optimizations, coherence in all its appearances (see Section 2.4.1 and Section 2.4.2) is the key factor for increasing ray tracing performance to a realtime level.

## 2.1 The Ray Tracing Algorithm

In order to avoid confusion, the term *ray tracing* or *core ray tracing* is defined here as an algorithm for finding the closest intersection between a ray and a set of geometric primitives, e.g. triangles [Glassner89, Badouel92, Erickson97, Möller97, Shoemake98, Shirley02, Wald04] or freeform patches [Sweeney86, Parker99b, Nishita90, Martin00, Wang01, Benthin04]. According to the ray equation $R(t) = O + t * D$, where $O$ is the ray origin and $D$ the ray direction, the ray tracing algorithm returns the intersection with the smallest distance $t_{min} \in [0, \infty)$. Other ray tracing algorithms that, for example, return all intersections along a ray will not be considered in this thesis. The smallest distance $t_{min} \geq 0$ corresponds to the closest intersection with respect to the ray origin. The actual intersection point can be easily obtained by evaluating

the ray equation using $t_{min}$. Note that the algorithm can easily be adapted so as to accept only intersections that lie in a given distance range $[t_{start}, t_{end}]$.

Using a brute force approach by testing all primitives within a scene against the ray and comparing the resulting intersection distances, is only an option if the number of primitives is small. Unfortunately, the typical number of primitives per scene ranges from thousands to millions making the brute force approach for realtime rendering impractical.

A well-known optimization technique consists in applying spatial subdivision. Spatial subdivision splits the virtual scene into spatial cells and stores for each cell the geometric primitives contained within. Note that only those primitives that are contained in the spatial cells intersected by the ray need to be tested. The advantage of this approach is that only a small subset of all spatial cells has to be considered for a given ray. Moreover, a typical scene includes large regions of empty space, so that for most rays only a small number of non-empty cells have to be considered. Exploiting spatial subdivision allows for significantly reducing the required primitive tests per ray, but also introduces the new problem of quickly identifying those cells that are intersected by the ray. The data structures required for the spatial sorting of geometry are called *spatial index* or *spatial acceleration structures*.

Algorithms for quickly identifying intersected cells are called *ray traversal algorithms*. Note that only those spatial cells have to be traversed which are pierced by the ray when starting from its origin and following the ray direction. Keeping to such a traversal "direction" allows for efficiently finding the closest intersection without touching any cells beyond that point. Exiting the traversal at the closest intersection is called *early ray termination*. Early ray termination can save many traversal and intersection steps (for the cells beyond the closest intersection point), but also requires a front-to-back traversal order of all cells that are intersected by the ray.

Researchers have proposed many different spatial index structures such as *bounding volume hierarchies* [Rubin80, Kay86, Haines91, Smits98], uniform, non-uniform, and hierarchical *grids* [Fujimoto86, Amanatides87, Gigante88, Jevans89, Hsiung92, Cohen94, Klimaszewski97], *octrees* [Glassner84, Samet89, Cohen94, Whang95], axis-aligned *BSP* (binary space partitioning) trees, short *kd-trees* [Sung92, Subramanian90a, Bittner99, Havran01], and ray direction sorting techniques such as *ray classification* [Arvo87, Simiakakis95]. These techniques either sort the primitives within a scene hierarchically (bounding volume hierarchies) or subdivide the space spanned by the primitives hierarchically (grids, octrees, kd-trees).

As a result, finding the closest intersection point between a ray and a set of geometric primitives includes the front-to-back traversal of a spatial index structure and the intersection tests for the corresponding primitives.

## 2.2 Ray Tracing for Rendering

The core ray tracing algorithm represents the fundamental basis for many ray tracing-based rendering algorithms. The common task of all these rendering algorithms is to compute a two-dimensional image from a three-dimensional virtual scene. As ray tracing is well suited for accurately simulating the distribution of light by simulating the propagation of photons, most ray tracing-based rendering algorithms focus on providing high image realism by closely simulating the illumination within the virtual scene. The realism within the generated image largely depends on how accurately the rendering algorithm takes into account both the illumination (by light sources) and the surface properties (of the scene's geometry). As a general rule, the higher the desired accuracy, the more rays have to be shot during simulation.

In order to give a better understanding of how rendering algorithms rely on the core ray tracing algorithm, a brief illustration of the standard *recursive ray tracing approach* [Whitted80] (see Figure 2.1) is presented next.

Generating a two-dimensional image from a three-dimensional scene using recursive ray tracing comprises several steps: From a virtual camera (typically corresponding to a one-eyed imaginary observer), rays are created and shot through each pixel of a virtual image plane. These rays are called *primary rays*, because they are created first. For each primary ray, the core ray tracing algorithm returns the closest intersection, in the following simply called *hit* or *intersection point*, with the geometry of the scene.

In order to determine the light at an intersection point that is reflected in the (inverse) direction of the ray, the illumination at a given point has to be computed first. Afterwards, the illumination must be combined with the material properties of the underlying geometry. The process of determining the interaction of light with the material properties at an intersection point is called *shading*. Note that the light reflected at the intersection points of primary rays determines the final pixel color.

The decision of whether a light source contributes illumination to a given hit point or not can be based on a simple occlusion test: A ray is shot from a hit point towards a virtual light source (or vice versa), and only if no intersection occurs along the way, the light source will contribute to the illumination. As these rays determine whether a given hit point lies in shadow or not, they are called *shadow rays*. For the occlusion test itself, the core ray tracing algorithm can be used once more, but it might be useful to slightly modify the algorithm. For shadow rays, the determination of the intersection with the smallest distance $t_{min} \in [0, distance(lightsource)]$ is not mandatory; instead, any intersection with a distance $t \in [0, distance(light\ source)]$ will be sufficient. This allows for more efficient ray termination because as soon

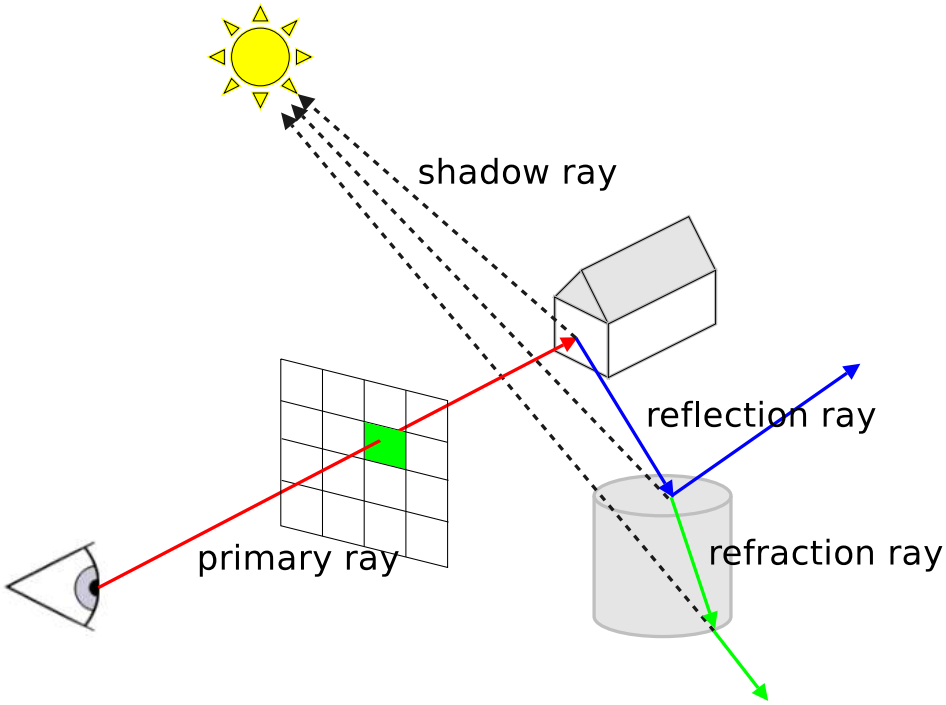



*Figure 2.1: Simplified illustration for recursive ray tracing: A primary ray is generated from a virtual camera and shot through each pixel of the image plane. The intersection closest to the ray origin is determined and tested for illumination by the light sources (the green shadow rays). Depending on the material properties of the intersected geometry and the incoming illumination, the current intersection point is shaded and potential reflection or refraction rays are generated. For each of these secondary rays, the contribution is recursively evaluated in the same way as for primary rays.*

as an appropriate intersection is found, the ray tracing algorithm can be terminated.

Based on the material properties of the geometry and the underlying rendering model, additional secondary rays can be generated to simulate effects such as reflection or refraction. Depending on their purpose, these rays are also called *reflection rays* or *refraction rays*. The contribution of secondary rays to the current hit point is recursively evaluated. Even though these rays are called secondary rays, they are essentially treated the same way as primary rays.

Apart from this traditional and simple approach to recursive ray tracing, countless variations exists for ray tracing-based rendering [Glassner89,

Glassner95, Shirley03]. For example, Cook extended the recursive ray tracing approach to support additional effects such as glossy reflection, illumination by area light sources, motion blur, and depth of field. This extended approach is called *distribution ray tracing* [Cook84a]. More advanced algorithms even compute the complete global illumination within a scene, including indirect illumination and caustic effects [Cook84b, Lafortune93, Cohen93, Veach94, Veach97, Jensen01, Shirley03, Dutre03]. Even though the purpose and supported accuracy of each algorithm is different, the key point is that they all heavily rely on the core ray tracing algorithm as their fundamental base.

## 2.3 Ray Tracing Performance

Looking at the organization of ray tracing-based rendering, it becomes clear that all algorithms require to trace a massive number of rays. Casting only a single primary ray per pixel at a resolution of $1024 \times 1024$ results in over one million rays per image. In particular, each ray must locate the correctly intersected primitive in a scene of (typically) millions of primitives. A performance level that allows for generating multiple images per second requires therefore a very fast implementation of the core ray tracing algorithm.

Section 2.1 illustrated that the core ray tracing algorithm basically consists of two coupled operations: traversal of a spatial index structure and ray primitive intersection tests. This is why the performance of the core ray tracing algorithm largely depends on the total number of required traversal and intersection operations. Obviously, the cost of these operations is closely related to their implementation in terms of the underlying hardware architecture.

If one has a closer look at the core ray tracing algorithm in terms of computational and memory-related operations, it becomes clear that a ray traversal step mainly involves data-dependent computations. Given the current traversal state, which corresponds to a location within a spatial index structure, a new state (location) is computed based on the ray data and the data loaded from the spatial index structure. As traversal states depend on data from the spatial index structure, a data dependency exists.

For a single intersection computation, the same data dependency applies (loading primitive data from memory first, then performing intersection computation). However, multiple intersection tests in a given spatial cell do not depend on each other, so that the loading of primitive data and intersection computation can be interleaved or even done in parallel.

The main problem of memory-related operations is that the data access latency might be the limiting factor. Prior to any traversal or intersection

operation, the required data has to be loaded from memory first. For an implementation on a current CPU architecture, this could become critical due to the discrepancy between the memory latency of on-chip cache and system memory (a more detailed discussion will be given in Chapter 3). Ensuring a large number of data accesses to the on-chip cache is essential to achieve high performance.

## 2.4 Coherence

Defining the term *coherence* is not a trivial matter, as its meaning depends to a great extent on the environment where it is applied. Nevertheless, the following definition illustrates the term *coherence* in a non-restricting way:

**Definition**: *Coherence* is the degree of which parts of an environment or its projections exhibit local similarities [Foley97].

Transferring the definition of coherence into the context of ray tracing, raises the question of which parts of ray tracing exhibits local similarities. In order to answer this question step by step, the search for coherence is first restricted to the core ray tracing algorithm. Later, the search is extended to the shading process which forms another major part of a ray tracing-based rendering system.

### 2.4.1 Ray Coherence

Defining *ray coherence* as the degree of spatial deviation within a set of rays, a statement on coherence in the context of traversal and primitive intersection becomes possible.

More precisely, a set of coherent rays (see Figure 2.2) will basically travel through the same spatial regions, and thus access the same data of the spatial index structure. The same holds true for the intersection test, where a set of coherent rays will basically test similar primitives. As a result, one can define *traversal coherence* for a set of rays as the ratio between the number of spatial cells traversed by all rays (as entity) and the sum of cells traversed by any ray. In the same manner, one can define *intersection coherence* for a set of rays as the ratio between the number of intersection tests performed by all rays (as entity) and the number of intersection tests performed by any rays.

The higher the traversal and intersection coherence the higher the probability of loading the same data, exploiting the memory cache hierarchy of current CPU architectures. In this case, traversal and intersection coherence translate into *memory coherence*.

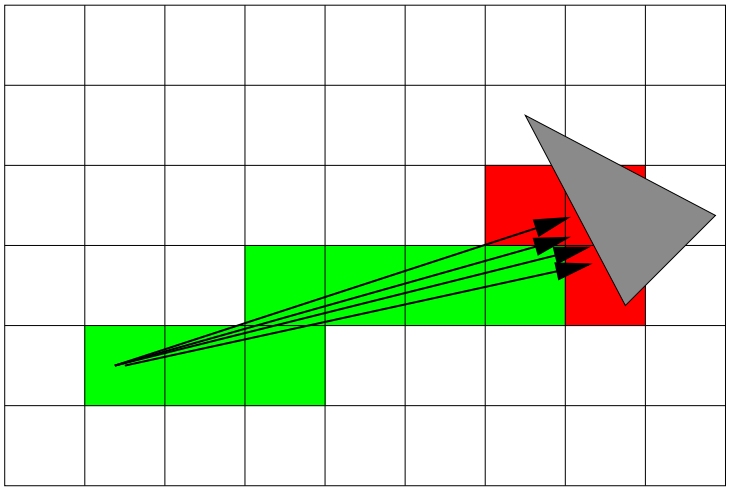


*Figure 2.2: Exploiting ray coherence for ray tracing: A set of coherent rays will essentially follow the same path through the spatial index structure (green cells = accessed by all rays, red cells = accessed by some rays), and will essentially test the same primitives for intersection. This allows for efficiently amortizing memory accesses over all rays in a set.*

Having data in the cache shortens the access latency for the individual rays within a set, but does not reduce the required number of memory accesses itself. A simple way of approaching this problem is a change in the way in which rays are handled. Instead of tracing rays individually, it is more beneficial to trace them together as a set.

As illustrated in Section 2.3, the core ray tracing algorithm involves many data-dependent computations and therefore numerous memory accesses. Coherent sets of rays allow for efficiently reducing the number of memory accesses due to the high probability that all rays within the set request the same memory (memory coherence), while the cost of the memory access itself is amortized over all rays within a set. As no dependency exists between the rays of a set, the traversal and intersection operations are suitable for parallel processing. In the following, a set of coherent rays is called a *coherent bundle*, *bundle* or *packet*.

Special attention must be given to the fact that an algorithmic or implementation-based overhead for supporting ray bundles does not offset the corresponding benefit. Therefore, the following three requirements must be fulfilled in order to fully exploit the advantages of coherent ray bundles for the core ray tracing algorithm:

**Spatial Index Structure:** The spatial index structure and the corresponding traversal algorithm must efficiently support ray bundles. The traversal algorithm in particular should avoid any complex computations.

**Ray Bundle Traversal:** An implementation of ray bundle traversal (with respect to a chosen spatial index structure) must minimize any overhead due to inefficient mapping to the underlying hardware architecture. Ideally, the hardware architecture and its corresponding instruction set should optimally support an implementation.

**Ray Bundle Intersection:** Similar to a traversal implementation, an intersection implementation has to efficiently support ray bundles. As different types of geometric primitives can exist in the same scene, ray bundle intersection algorithms should efficiently support various primitive types.

In the case no efficient ray bundle intersection algorithm can be found for a given primitive type, the fall-back solution of intersecting the rays sequentially might be acceptable.

Chapters 4, 5, and 6 will provide a detailed discussion of data structures, algorithms, and the corresponding implementations that fulfill all these requirements.

### 2.4.2 Shading Coherence

Besides the actual core ray tracing algorithm, the shading process also benefits from local similarities. Considering for example the hit points of neighboring primary rays which are likely to intersect the same primitive. Even if this is not the case, it is very likely that the hit points are in the same spatial region. In addition, the probability that the intersection points share the same shader, and therefore perform similar shading operations is very high. Similar shading operations could be performed in parallel, again allowing for processing the shading of intersection points in bundles.

Sharing the same shader implies additionally that the loaded data exhibits similarities as well. For example, look-ups to the same texture for neighboring intersection points are likely to provide a high degree of memory coherence, which in turn allows for the efficient use of caches.

As a result, the shading process itself can benefit from coherence; however, similar to the core ray tracing algorithm, bundle shading for intersection points is prone to overhead caused by inefficient implementation.

## 2.5 Conclusions

This chapter has demonstrated that the key to pushing ray tracing performance to a realtime level is an efficient support of coherent ray bundles combined with a highly optimized implementation. A sub-optimal implementation in particular is likely to offset any benefit offered by tracing bundles. Unfortunately, current CPU architectures in combination with their supporting compilers are not intended to support the required degree of efficiency out of the box. On the contrary, without any manual effort in optimizing algorithms and implementation code, there is only little chance of ever reaching the desired performance level.

In order to avoid hardware and software-specific implementation pitfalls, it is beneficial to examine the architecture of current CPUs more closely (see Chapter 3). Taking specific CPU-related issues into account allows for an efficient implementation of the core ray tracing algorithm. In particular, the optimized implementation includes the extension of the traversal algorithm (see Chapter 4) and the primitive intersection tests (see Chapter 5 and Chapter 6) for efficiently supporting ray bundles.

A difficult task for a ray tracing-based rendering system is to gather coherent ray bundles. Coherent bundles of primary rays can easily be generated when relying on a typical (perspective) camera model, where primary rays of neighboring pixels exhibit a large degree of coherence. The same holds true for the shadow rays generated towards a single point on a light source. However, for most secondary rays, e.g. those generated by reflection off of curved surfaces and even refraction rays, coherence will be significantly lower than for primary rays. In the case of low coherence, coherence-based regrouping or even falling back to tracing individual rays may be necessary.

Instead of trying to extract coherent ray bundles out of a traditional ray tracing-based rendering system, one can design the rendering system in such a way that the majority of rays can be combined directly in coherent bundles (see Chapter 9). Such a system allows for minimizing the overhead for generating coherent bundles. If for such a system the majority of rendering time is spent in the core ray tracing algorithm, total system performance will directly benefit from fast ray bundle tracing.

# Chapter 3
# CPU Architectures

Optimizing program code basically consists of two steps: Identifying the program's hot spots and trying to modify the corresponding code sequences. The difficulty thereby is to modify the code in such a way that the modified version shows better run-time behavior (with respect to the underlying hardware architecture) than the original one. This requires identifying code structures that cannot be efficiently executed. Identifying and modifying makes code optimization a difficult and time-consuming task which requires in-depth knowledge of the underlying architecture. Apart from architectural issues, code optimization additionally depends on the chosen compiler.

Nevertheless, the benefits of code optimization can be tremendous. Statistics have shown that the increase in performance between optimized and non-optimized code can range from a few percent to entire orders of magnitude.

In order to avoid costly performance penalties in critical code sequences, it is essential to have a detailed knowledge of the execution flow within the underlying CPU architecture. Therefore, a brief overview of performance-related issues of today's CPU architectures will be given in Section 3.1. Having identified the architecture related issues allows for formulating the general coding guidelines of Section 3.2. Section 3.3 introduces the concept of SIMD instructions, while Section 3.4 discusses compiler and performance profiling-related tools.

## 3.1 Performance Issues

For historical reasons, the majority of today's software has been designed for non-parallel execution. Because of this, current CPU architectures are designed to execute serial program code as fast a possible. With every new CPU generation, designers have tried to achieve a continuous performance

increase by introducing small architectural enhancements to optimize serial execution. The key factor to increase performance has been the raising of the CPU clock rate (up to 3.8 GHz for latest Pentium-IV [Intel01]). Having a higher clock rate allows for (potentially) executing more instructions within a fixed time period. Raising the clock rate could only be realized through smaller micro structure designs which require extremely long executing pipelines (the Pentium-IV uses a pipeline with more than 30 stages).

These long executing pipelines are the major drawback of the high clock rate architectures. If the utilization of the execution pipeline is low, the performance of the CPU itself will be low, too. Even though the analysis and optimization of execution bottlenecks is a quite complex topic, the main reasons for low pipeline utilization, and thus low performance, can be roughly classified into the following categories:

**Cache Misses:** The difference in access latency between on-chip memory cache and main memory itself is tremendous, e.g. the Pentium-IV has an L1 cache latency of 1-4 cycles, an L2 cache latency of 20-27 cycles, and a latency to main memory of over 200-300 cycles. Obviously, this can result in pipeline stalls whenever data does not reside in the memory cache hierarchy. CPU architectures follow multiple ways of reducing the impact of cache misses by having large and multiple cache levels to reduce the probability of cache misses, and by executing independent program code during idle periods. For serial program code, the search for, and the execution of, independent instructions is often referred to as out-of-order execution. On the other hand, techniques such as simultaneous multi-threading or hyper-threading [Intel02c] use the implicit parallelism of thread execution to fill pipeline stalls with independent instructions.

**Branches:** The negative impact of long executing pipelines comes into play when dealing with conditional branches. Every time a branch is wrongly predicted, the complete pipeline has to be flushed. Branch prediction (over multiple levels) and even branch prediction by branch hint instructions are able to lower the probability. Nevertheless, every mispredicted branch means a "waste" of up to 30 cycles for a 30-stage execution pipeline. Even though the negative performance impact is lower than for cache misses it is still significant. This particularly affects complex and branch-intensive code.

**Low Instruction Level Parallelism:** Multiple instruction pipelines and functional units combined with out-of-order execution even allow the serially operating CPU to execute multiple independent instructions in

| Processor | Intel Pentium-IV | AMD Opteron |
|---|---|---|
| Pipeline Stages | 20-31 | 12-17 |
| Reorder Buffer | 126 Entries | 72 Entries |
| Units | 2 Int, 1 FP, 3 LS | 3 Int, 3 FP, 2 LS |
| SIMD | MMX, SSE, SSE2, SSE3 | MMX, SSE, SSE2, SSE3 |
| Register | 16 Int, 8 FP, 16 SIMD | 16 Int, 8 FP, 16 SIMD |
| Caches (I, D, L2) | 12k $\mu$OPs, 8-16k, 512k-2M | 64k, 64k, 1M |
| Branch Prediction | 4k BTB | 16k BHT + 2k BTB |
| Memory Bandwidth | 6.4 GB/s | 6.4 GB/s + 3x HyperTP (3.2 GB/s) |
| Frequency | 1.5-3.8 GHz | 1.4-2.6 GHz |

*Table 3.1: Specifications of current 64-bit processor architectures, e.g. Intel's Pentium-IV (EM64T) and AMD's Opteron (X64). Both architectures feature a high pipeline length (varying due to different lengths for integer and floating pipelines) and large caches. Both architectures include special hardware units, such as Branch Target Buffers (BTB) and Branch History Tables (BHT), to achieve a better branch prediction rate. Large caches and advanced branch prediction units are the most effective components (of these architectures) to keep the long pipelines busy.*

parallel (within a single clock cycle). This feature is typically called *instruction level parallelism* (ILP). Code dependencies appear when an instruction requires the output of a preceding instruction as input, while the output is not available at this point (mostly due to instruction latency). A sequence of dependent instructions is called a *dependency chain*. Dependency chains frequently appear in inefficient and complex code, but long latency instructions such as divisions, function calls, operating system-related or input/output instructions can also contribute to these chains.

In order to lower the effect of the three above-mentioned bottleneck categories, the most recent CPU architectures feature large on-chip caches and special hardware components to keep pipeline utilization high (see Table 3.1). Unfortunately, the improved hardware support is not able to overcome the effects of inefficiently written code. Even though modern compilers [Intel02a, GNU] try to rearrange the instructions in a more suitable way, the optimum in run-time performance can only be achieved by optimizing data structures and algorithms manually to the underlying hardware architecture.

The road map of the leading CPU manufacturers makes it clear that the length of the pipeline, and thus the frequency of future CPUs is not likely to grow significantly, and may even decrease. Therefore, future CPU designs will rely on increased execution parallelism such as multiple pipelines and functional units, execution threads, and in particular many cores in order to increase performance. Introducing multiple cores will widen the gap between memory latency/bandwidth and compute power dramatically. As each core will access the main memory via the same memory interconnect, future algorithms will need to reduce their required memory bandwidth.

## 3.2 Coding Guidelines

The performance impact of inefficiently written code ranges from a few percent to entire orders of magnitude, largely depending on the code itself. Even though the following coding guidelines seem obvious, it is worth outlining them to reduce the probability of significant performance penalties:

**Data Locality and Memory Access Pattern:** Localizing memory access ensures high cache hit probability. In situations where large chunks of memory need to be loaded, a qualified memory access pattern such as pure sequential access should be applied. Sequential access is efficiently supported by the hardware prefetching unit, available on all current CPU architectures. In certain situations, it is beneficial to manually apply CPU-supported *prefetch* instructions in order to load chunks of memory in advance. As memory prefetching is done asynchronously to the execution flow, the loading latency can be hidden by working on data already residing in the cache hierarchy. Obviously, the mechanism only works if the memory loading latency is smaller than the computation time between subsequent prefetches.

**Simple Control Flow:** Replacing branch-frequent code by conditional move sequences avoids mis-predicted branches. However, if the probability of a branch mis-prediction is sufficiently low (something which depends largely on the code), applying branches can be faster than applying conditional moves, because the execution of a correctly predicted branch has almost no costs. In contrast, conditional move sequences may require the computation of results which may not even be used later. On a more higher level, critical run-time loops should be kept as small and as simple as possible. Simply dropping or replacing complex instructions such as divisions or functions calls in inner loops can dramatically speed up performance. Having only a minimum of code dependency

chains within the inner-most loop body can additionally increase instruction throughput. Furthermore, small and simple inner-loops can be more easily optimized and maintained.

**Data Level Parallelism:** In order to maximize performance, data level parallelism by SIMD (single instruction multiple data) instructions should be applied. SIMD instructions offer a simple and easy way to manipulate multiple data elements at once. In particular, Intel's SSE instruction set [Intel01, Intel02b, Intel02a] allows for manipulating four single precision floating point or integer values using a single instruction. The main issues when using SIMD instructions are the required layout changes of input data and the necessary recoding of the algorithm (see Section 3.3).

Data level parallelism via SIMD instructions is currently the only way of explicitly performing parallel operations on a sequentially operating CPU (see Section 3.3). Instruction level parallelism, on the other hand, is performed internally by the CPU (out-of-order execution). It can only be influenced implicitly by reordering code for a more appropriate out-of-order executing flow.

In contrast to single-threaded applications, coding guidelines for multi-threaded applications should also contain guidelines for thread synchronization. In order to efficiently exploit multiple CPUs (or CPU cores), synchronization points (such as mutex lock/unlocks [Nichols96]) should be kept to a minimum. Roughly speaking, synchronization means serialization, and long serialization periods dramatically reduce the positive effect of parallel execution.

Multi-threaded performance is further affected whenever *implicit synchronization* is required. Implicit synchronization occurs when caches on different CPUs need to be synchronized. Even though the interconnects between CPUs offer a high bandwidth for synchronizing cache areas, the performance impact can still be very high. Due to the fact that the synchronization granularity is based on cache-lines, unintended cache synchronization can be avoided by assuring that synchronization-relevant data exclusively occupies complete cache-lines.

## 3.3 Data Level Parallelism by SIMD Instructions

The basic concept behind SIMD instructions is the idea that many algorithms (see Figure 3.1) could work on multiple data elements in parallel. Instead of processing each data element by a single instruction in a sequential manner,

```
// the four three-component input vectors are
// stored in the SOA format.
//
// float x[4] -> the four 'x' vector components
// float y[4] -> the four 'y' vector components
// float z[4] -> the four 'z' vector components

inline void SimpleDot4(const float *x,
                       const float *y,
                       const float *z,
                       float *const dest)
{
 dest[0] = x[0]*x[0]+y[0]*y[0]+z[0]*z[0];
 dest[1] = x[1]*x[1]+y[1]*y[1]+z[1]*z[1];
 dest[2] = x[2]*x[2]+y[2]*y[2]+z[2]*z[2];
 dest[3] = x[3]*x[3]+y[3]*y[3]+z[3]*z[3];
}
```

*Figure 3.1: Computing a dot product of four three-component floating point vectors with standard C/C++ code. As no dependencies exists, the computation (for each component of 'dest') can be performed in parallel. Note that the code does not implement a generic dot product, but rather a simplified version where only one input vector is used for both dot product elements.*

one could apply the same operation to multiple elements in parallel using a single instruction.

Applying SIMD requires that no dependencies exist between the data elements in a processing group. Moreover, controlling program execution by conditional branches brings about certain changes: Instead of basing the branch on the result of a single condition, SIMD instructions compute the results of multiple conditions in parallel. Obviously, one could extract the result of a single condition and perform the branch with respect to it, but very often it is more beneficial to perform the branch based on a condition regarding all results.

SIMD features are available on almost all current processor architectures [Intel02b, AMD03, AltiVec, IBM05], but only *Intel's Streaming SIMD Extension* will be considered in the following. Nevertheless, all code examples in this thesis can be easily ported to a different SIMD extension.

### 3.3.1 Intel's Streaming SIMD Extension (SSE)

For historical reasons, floating point operations on x86 architectures have been executed using the FPU (floating point unit). FPU registers have been

organized in a stack-like structure, causing reorganization overhead for executing floating point operations. With the introduction of the Pentium-III, Intel for the first time offered SIMD features with its Streaming SIMD Extension (SSE) [Intel02b].

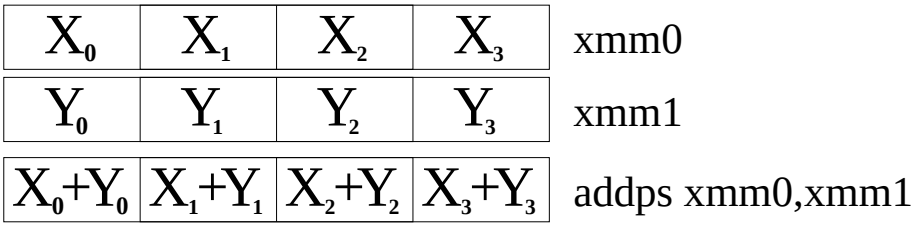



*Figure 3.2: Intel's streaming SIMD extension allows for manipulating four data elements via a single instruction, e.g. adding four single precision floating point values.*

SSE allows for operating on multiple data elements, usually on four 32-bit data elements, for the cost of one. Besides the advantage of data level parallelism, dropping the stack-like register organization in favor of a linear addressing model further accelerates operations further.

The standard SSE instruction set allows for operating on four single precision floating point values via a single instruction. SSE implementations of current processors, e.g. Pentium-IV internally execute only two single precision floating point operations in parallel, i.e. four operations in a $2 \times 2$ way. This hardware implementation is currently limiting the SSE instruction throughput, but is likely to be removed from future architectures [Intel05].

The latest extension to the instruction set, called SSE2 [Intel01] and SSE3 [Intel03], also makes it possible to manipulate four 32-bit integer, eight 16-bit integer, sixteen 8-bit integer, or two 64-bit double precision floating point values via a single instruction. As these instruction do not play a major role in the example implementations, they are not discussed in any more detail.

All SSE registers (8 to 16, depending on the architecture version) are $4 * 32 = 128$-bit wide. These registers are usually referenced by *xmm0* to *xmm7* (or *xmm15*). Figure 3.2 shows a simple example of adding two SSE registers (four single precision floating point values per register) using a single instruction.

SSE introductions work most effectively in the vertical way, as shown by Figure 3.2. The horizontal way is more difficult due to the lack of appropriate horizontal instructions. For example, a short sequence of instructions is required to sum up all four register elements (see Section 3.3.3). Even though

the latest architectures [Intel03] offer direct support for horizontal operations, they are not as efficient implemented as vertical operating instructions.

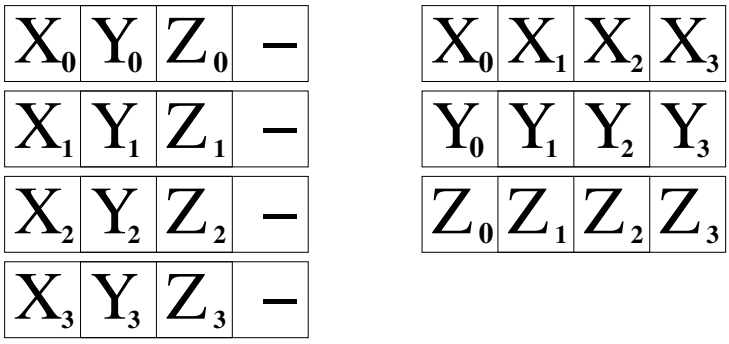



*Figure 3.3: Left: For many algorithms, e.g. computing dot products, the standard array-of-structure (AOS) layout does not ensure optimal SSE utilization (the right-most element is not used). Right: The structure-of-array (SOA) layout is more beneficial as it optimally supports vertically operating SSE instructions.*

As a consequence, the layout of data and the implementation of algorithms should be rearranged to allow for as many vertical instructions as possible. As an example, vertex data should be stored in the *structure-of-array* (*SOA*) format instead of the *array-of-structure* (*AOS*) format (see Figure 3.3). Figure 3.4 shows that using the structure-of-array data layout, four dot products can be efficiently implemented by using only five instructions. The array-of-structure data layout would require more than double the number of instructions (because of the required shuffle operations).

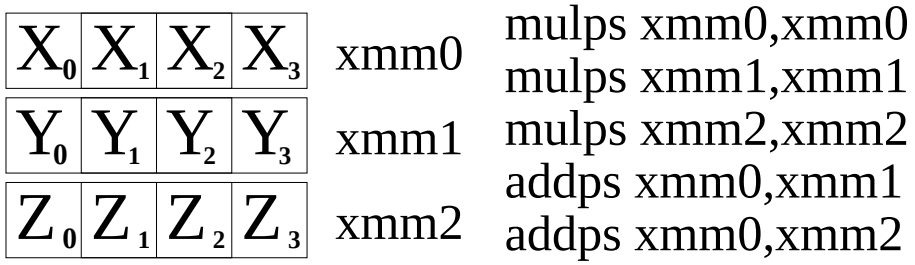



*Figure 3.4: The* structure-of-array *(*SOA*) data layout allows for efficiently using SSE instructions, e.g. computing four (simple) dot products in parallel (similar to the code in Figure 3.1) using only 5 instructions.*

If the array-of-structure layout as input format cannot be avoided, a corresponding transposition into the structure-of-array can be performed on-the-fly. The transposition code consists of a sequence of shuffle and load instructions. The decision of whether the overhead introduced by the rearrangement of register elements is tolerable or not largely depends on the code. If the code following the transposition is sufficiently long, the introduced overhead can be effectively amortized.

Another important factor concerning efficient coding with SSE instructions relates to the rearrangement of instructions in terms of their throughput and latency. In particular, conditional branches based on a comparison between SSE registers require a manual transfer of the resulting bit mask (see Section 3.3.2) from an SSE register to a general purpose register. This transfer suffers from an high latency and should therefore be used carefully. On the other hand, coding in terms of instruction throughput is essential for avoiding resource conflicts of computational units. Throughput and latency vary across x86 architectures (and their variants), making general coding optimizations difficult. Fortunately, modern compilers allow for reordering code with respect to a specific processor without much manual effort.

Another important limitation of SSE instructions is the lack of scatter/gather operations for loading and storing data. Loading or storing individual register elements requires a sequence of loading and shuffle operations, because only the lowest register element can be handled as a standard scalar register.

### 3.3.2 SSE Intrinsics

One possible way to benefit from SSE instructions is to directly rely on assembly programming, but this procedure would be too inflexible and error-prone. On the other hand, modern C/C++ compilers (see Section 3.4) offer auto-vectorization of standard C/C++ program code. Even though this allows for programming in standard C/C++ code, the output typically will not reach the quality of hand-written code. Furthermore, the compilers often fail to auto-vectorize loops because of unresolvable dependencies (from the compiler's view) within the loop body. A more efficient way that combines both the quality of hand-written assembly code and the high-level interface of C/C++ code are SSE *intrinsics* [Intel02a]. Intrinsics are C/C++ function-style macros, which can be used directly with C/C++ constructs and variables. During compilation, the compiler automatically takes care of register allocation, result propagation, intermediate usage, etc.

Due to the fact that the compiler takes care of SSE register and memory allocation, the programmer can focus on the algorithmic implementation

```
// the four three-component input vectors are stored
// in the SOA format using a SIMD vector data type.
//
// u[0] -> the four 'x' vector components
// u[1] -> the four 'y' vector components
// u[2] -> the four 'z' vector components

typedef typedef __m128 sse_t;

inline sse_t SimpleDot4SSE(const sse_t *u)
{
  return _mm_add_ps(_mm_add_ps(_mm_mul_ps(u[0],u[0]),
                               _mm_mul_ps(u[1],u[1])),
                    _mm_mul_ps(u[2],u[2]));
}
```

*Figure 3.5: Intrinsics allow for effectively using SIMD instructions with C/C++ constructs. This small routine computes four (simple) dot products (similar to the code in Figure 3.1) in parallel using SSE intrinsics.*

itself (see Figure 3.5). Moreover, the compiler is able to perform architecture-specific optimizations, e.g. considering throughput and latency optimizations. Most SSE intrinsics map to a single SSE instruction, others are composed of a short sequence of SSE instructions.

Given the importance of SSE instructions for efficient coding, the most frequently used instructions will be discussed in more detail. Note that scalar operating intrinsics, operating only on the lowest register element, differ from their full counterpart only through the *_ss*-suffix instead of the *_ps*-suffix.

**Arithmetic Instructions** For addition, multiplication, subtraction, and division of four single precision floating point values, the intrinsics *_mm_add_ps*, *_mm_mul_ps*, *_mm_sub_ps*, *_mm_div_ps* are used. Reciprocal operations are also useful, e.g. *_mm_rcp_ps* and *_mm_rsqrt_ps*, which perform the $1/x$ and $1/sqrt(x)$ operations. These special instructions are faster than a real division or square root. The speed increase comes at the expense of reduced accuracy because the result is computed based on approximating algorithms. Nevertheless, the reduced accuracy can be increased afterwards by performing a Newton-Raphson iteration [Intel02b] (resulting in a short sequence of additional instructions). Breaking up costly long-latency instructions such as real divisions or square roots into a short sequence of short-latency instructions is more advantageous for exploiting instruction level parallelism.

**Load/Store Instructions** Loading one single or four single precision floating point values can be performed using the intrinsics *_mm_load_ss* and *_mm_load_ps*. Note that the argument of *_mm_load_ss* and *_mm_load_ps* is a pointer to a float or float array. Additionally, *_mm_load_ps* requires that the address is aligned on a 16-byte boundary to ensure maximum performance (less efficient non-aligned access can be implemented by *_mm_loadu_ps*). The intrinsics *_mm_set_ps* and *_mm_set_ps1* use immediate float values as arguments. *_mm_set_ps1* copies a single value into all four register elements. Note that loading four different float values into a single register by applying *_mm_set_ps* and *_mm_rset_ps* involves a sequence of shuffle instructions. Returning the lowest SSE register element as scalar float value can be performed by using the *_mm_cvtss_f32* intrinsic.

**Comparison Instructions** Comparisons between two SSE registers using the *_mm_cmpXX_ps* (XX can be *eq*,*lt*,*gt*,...) return a mask where all bits of each register element are set to one if the result of the comparison is true and are set to zero otherwise. In combination with a sequence of logical instructions, the returned bit mask can be used to implement an element-based conditional move sequence.

**Miscellaneous Instructions** The *_mm_shuffle_ps* intrinsic shuffles register elements within one single or across two registers (with certain restrictions). It is often necessary to apply a conditional jump based on the values of all four register elements. Therefore, *_mm_movemask_ps* transfers the highest bit (the sign bit) of each register element to the four lowest bits of a general purpose register. If all register elements contain a comparison's bit mask, the transferred bits correspond to the comparison result. In order to quickly obtain the maximum and minimum values per register element, the *_mm_max_ps* and *_mm_min_ps* intrinsics can be used.

**Logical Instructions** The logical operations *or*, *and*, *xor*, and *notand* are realized by *_mm_or_ps*, *_mm_and_ps*, *_mm_xor_ps*, and *_mm_andnot_ps*. *_mm_setzero_ps* allows for quickly setting all register elements to zero using a simple *xor* operation.

### 3.3.3 Using Intrinsics and Basic Data Structures

As discussed in Section 3.2, the performance impact of mis-predicted branches due to pipeline flushes can be critical. This holds especially true for SSE code sequences. In many cases, a conditional branch construct looks like:

IF *condition* THEN *C=A* ELSE *C=B*. An efficient way to realize these if-else-constructs per SSE register element without branching is to apply bit masking (returned by an SSE-comparison) using a short sequence of logical instructions (see Figure 3.6).

```
inline sse_t Update4(const sse_t a, const sse_t b, const sse_t mask)
{
  // note that 'andnot' requires the 'mask' as first parameter
  return _mm_or_ps(_mm_and_ps(a,mask),
                   _mm_andnot_ps(mask,b));
}
```

*Figure 3.6: Small utility functions increase the readability of SSE code significantly. The routine implements an - IF condition THEN C=A ELSE C=B - construct on register element level.*

For the operation of computing the inverse of four single precision values in parallel, it is often beneficial to not apply a real division, but to use a reciprocal approximation combined with a Newton-Raphson iteration in order to increase the result's accuracy later on. Figure 3.7 shows the actual SSE implementation.

```
inline sse_t Inverse4(const sse_t n)
{
  const sse_t rcp = _mm_rcp_ps(n);
  return _mm_sub_ps(_mm_add_ps(rcp,rcp),
                    _mm_mul_ps(_mm_mul_ps(rcp,rcp),n));
}
```

*Figure 3.7: Fast inverse computation by using reciprocal approximation combined with a Newton-Raphson step in order to increase IEEE precision to 24 bits. In many cases, splitting up inverse computation into multiple instructions is faster than using an explicit division, because the instruction level parallelism of current CPUs allows for a efficient parallel execution with surrounding instructions.*

Up to the latest Pentium-IV [Intel03] and Opteron processor generations, SSE instructions have been lacking direct hardware support for horizontal operations. Horizontal operations work on all elements of a single register and store the result in the lowest element. However, horizontal operations can be simulated using a sequence of instructions. As shown in Figure 3.8, a *horizontal add* requires at least four instructions. Similar utility functions compute horizontal *sub*, *min*, and *max* operations.

```
inline sse_t sseHorizontalAdd(const sse_t &a)
{
  const sse_t ftemp = _mm_add_ps(a, _mm_movehl_ps(a, a));
  return _mm_add_ss(ftemp,_mm_shuffle_ps(ftemp, ftemp, 1));
}
```

*Figure 3.8: Many architectures do not offer SSE instructions for performing horizontal operations, e.g. adding all four register elements and storing the result in the lowest element. Instead, a short sequence of instructions emulates these horizontal operations.*

Given the seamless integration of intrinsics into C/C++ code, it is beneficial to implement useful code sequence as small utility functions. These utility functions increase the readability of the SSE code significantly.

Another important point to consider is the fact that even coherence is highly important for implementing algorithms through SSE code. This is especially true for conditional branching based on the values of all four register elements (using the *_mm_movemask_ps* instruction). In this case, *decision-coherence* which is defined as the probability that the comparison result in all four register elements is equal, is essential. Testing the sixteen possibilities by *if-else* or *switch* statements, where each case has equal probability, would typically result in many branch mis-predictions. Therefore, a high decision-coherence allows for arranging the comparison statements based on their probability. This efficiently reduces the amount of mis-predicted conditional branches.

Creating fundamental data structures for tracing ray bundles is straightforward. Due to the fact that SSE operates on four elements at once, structures for storing information of four rays are created first. Figure 3.9 shows data structures for storing four three-component vectors in the SOA format, and the hence constructed four-ray structure.

Besides a structure for ray and intersection data, a structure for axis-aligned bounding boxes, e.g. as representation for spatial voxels, is often required. An efficient representation based on the extremal expansion is shown in Figure 3.10.

## 3.4 Tools and Hardware

Because of the required SSE intrinsics support, all algorithms were implemented using either the GCC compiler collection [GNU] or the Intel C/C++ compiler suite [Intel02a]. Because of its superior performance, the Intel com-

```
struct R3 {
  float x,y,z;
};

// -- single ray --
struct Ray {
  R3 direction;
  R3 origin;
};
struct Intersection {
  float dist;
  float u;
  float v;
  float id;
};

// -- packet/bundle of four rays --
struct SSEVec4 {
  sse_t t[3]; // SOA format
};
struct SSERay4 {
  SSEVec4 direction;
  SSEVec4 origin;
};
struct SSEIntersection4 {
  sse_t dist;
  sse_t u;
  sse_t v;
  int id[4];
};
```

*Figure 3.9: 'R3' is the basic representation of a three-component vector, while 'Ray' and 'Intersection' store ray and intersection data for a single ray. 'SSEVec4' is the fundamental data structure for storing four three-component vectors in the SOA format. It builds the base for the 'SSERay4' structure, which stores the data for four rays. The 'SSEIntersection4' structure stores the data for four intersection points, e.g. intersection distance, barycentric coordinates, and the index of the primitive hit.*

piler is favored over the standard GCC compiler. It is interesting to see that the Intel compiler creates very efficient code not only for Intel processors, but also for its competitors.

For profiling, analyzing, and identifying code bottlenecks, the VTune profiling tool is used [Intel04]. The tool provides a detailed analysis for detecting performance-related bottlenecks caused by cache misses, branch

```
struct Box {
  R3 min;
  float dummy0;
  R3 max;
  float dummy1;
};
```

*Figure 3.10: Structures for representing three-component vectors and axis-aligned bounding boxes. The axis-aligned bounding box is represented by their extremal points. The additional dummy values 'dummy0' and 'dummy1' increase the total structure to a preferable size of 32 bytes, which is more suitable with respect to cache-line sizes. Note that no SSE data types are used because for many algorithms a fast access to individual vector elements must be ensured (avoiding the overhead for accessing SSE register elements). In the two dummy values additional information can be stored.*

mis-predictions, etc. Furthermore, VTune provides many useful statistics such as cycles per instructions ratios, issued/retired ratios with respect to different instruction types, etc.

For providing an approximate cost measure for a short code sequence, the internal processor's clock cycle counter [Intel97] is used. Note that out-of-order execution, caching effects, varying branch table history entries, surrounding code sequences, and even differences in the underlying architecture make it very difficult to determine the exact amount of cycles required for a given code sequence. Therefore, all cycle statistics in this thesis should be seen as estimates rather than exact values.

As operating system a 32-bit or 64-bit Linux distribution, running a 2.4 kernel has been used. Most of the performance statistics were measured on a 2.2 GHz Pentium-IV PC with 512 MB of memory. In a few cases, in particular where a PC cluster setup was needed, a cluster of PCs with AthlonMP 1800+ PCs (running at 1.5 GHz) was used. Whenever a 64-bit processor provided superior performance compared to a 32-bit processor, a 64-bit AMD Opteron PC with 1.8 GHz has been chosen.

## 3.5 Conclusions

Current CPU architectures are able to provide a large amount of computing power if they execute well-suited code. Even though today's compilers offer automatic optimization features, the full potential of a processor can only be utilized by manual optimization.

The difference in performance between manually optimized and unoptimized code can reach order of magnitudes. Therefore, it will become more and more important to focus on the optimization of critical code sections. It can even be beneficial to favor algorithms that have a sub-optimal algorithmic complexity, but allow for very simple and therefore fast implementation. An implementation requiring only a sequence of simple instructions is typically more suited for current CPU architectures.

Apart from simplicity of implementation, the memory access pattern of algorithms is another critical point. Code that is optimized to access memory in a cache-friendly way is likely to outperform unoptimized code. The same optimization argument holds true for branch and ILP optimizations.

As a result, it is essential for achieving optimal performance on current CPU architectures to carefully chose and optimize algorithms in terms of the underlying architecture.

# Chapter 4
# Tracing Coherent Ray Bundles

Efficiently searching for the first intersection along a ray requires building up a spatial index structure over the primitives within the scene. The spatial cells created by the spatial index structure allow for efficiently reducing the number of potential intersection candidates but require a way of quickly identifying those cells that are intersected by the ray. Moreover, the cells have to be traversed in front-to-back order to ensure early ray termination.

The spatial index structure has a tremendous impact on the performance of the core ray tracing algorithm. The following issues describe those impacts in detail:

**Quality:** The quality of a spatial index structure is measured in terms of the number of required traversal steps versus resulting intersection operations. Depending on the chosen index structure and the corresponding implementation, it can be beneficial to increase the required number of operations of one category in order to decrease the number of operations of the other category, i.e. to have a higher number of traversal operations in order to perform fewer intersection operations. Nevertheless, the main goal should be to reduce the number of operations in both categories.

**Memory Layout:** The larger the memory representation of a spatial index structure the higher the required bandwidth to memory. In the case the processor's cache hierarchy is not able to efficiently support memory accesses to the spatial index structure, the core ray tracing algorithm can easily become memory-bandwidth limited. Moreover, a sub-optimal memory access pattern (e.g. accessing data crossing cache-lines) or a too complex data extraction sequence (e.g. decompressing spatial cell data) can significantly impact performance.

**Traversal Costs:** Besides the number of traversal steps, the cost of performing a single traversal step for one or more rays is particularly important. Complex code that does not efficiently map to the underlying hardware architecture is able to offset any benefit from a high-quality spatial index structure.

Apart from quality and memory requirements, the traversal operation must be generalizable for efficiently supporting coherent ray bundles (see Section 2.4.1), because this has turned out to be one of the key factors for pushing ray tracing performance [Wald04, Reshetov05]. This means that data access costs can be efficiently amortized over all rays within the bundle (instead of only one ray at a time). This allows for reducing the required memory bandwidth, which is known to be one of the major bottlenecks of current CPU architectures. Since a traversal operation is performed in a similar way for all rays within the bundle, an implementation can reduce traversal costs by exploiting SIMD instructions.

By carefully reviewing different spatial index structures with respect to quality, memory layout, and traversal costs, a kd-tree is chosen as the most appropriate data structure. A kd-tree is a BSP tree [Subramanian90a, Sung92, Bittner99, Havran01] with axis-aligned splitting planes. The advantages of a kd-tree are its better geometric adaptation ability [Havran01, Wald04] compared to other spatial index structures [Havran01], and its very efficient memory layout with respect to CPU caches [Wald04]. Moreover, kd-tree traversal can be efficiently generalized for tracing bundles of coherent rays in parallel using a short and iterative approach [Wald04, Reshetov05].

As algorithms for tracing coherent ray bundles form the crucial base for achieving realtime ray tracing for different primitive types (see Chapter 5 and Chapter 6), a detailed algorithmic review and coding guidelines for different implementations will be presented in the following. Section 4.1 briefly deals with an efficient memory layout for kd-trees, while Section 4.2 and Section 4.3 present state-of-the-art traversal algorithms for ray bundles in detail. Section 4.4 finally discusses the advantages and disadvantages of using ray bundles.

## 4.1 kd-Trees

The kd-tree is basically a binary tree in which each node corresponds to a spatial cell. Inner nodes within the tree also represent axis-aligned splitting planes. The spatial cells of the two child nodes of a given node are created by splitting the node's spatial cell by the corresponding splitting plane. The

deeper a node is located in the tree the smaller its spatial cell. Instead of splitting planes, leaf nodes store references to all primitives that intersect the corresponding spatial cell.

The kd-tree itself is constructed by recursively subdividing the scene's spatial bounding box by inserting axis-aligned splitting planes. As an inner node represents a splitting plane, the two child nodes represent two axis-aligned spatial cells, split by the splitting plane of the parent node. These spatial cells are often referred to as *cells* or *voxels*.

Each leaf node, in contrast to an inner node within the kd-tree, must refer to a corresponding list of primitives (instead of referring to the two child nodes). In other words, the representation of a kd-tree node requires a flag in order to decide whether it is an inner or a leaf node. In the event of an inner node, a pointer to the two child nodes (storing the child nodes successively saves a second pointer) is required, while in the event of a leaf node, a pointer to a list of primitive references is needed. Instead of using pointers, address offsets can be used. Adding the offset to the memory address of a given node yields the memory address of the two child nodes.

A compact data structure for realizing these requirements has been proposed by Wald et al. [Wald04]. Figure 4.1 shows that the 8-byte structure allows for efficiently storing all required information within a 64-bit word. Note that logical instructions such as *and*, *or*, and *xor* on 32-bit or 64-bit integers are very efficient on current CPU architectures, allowing masking operations to extract data to be implemented very efficiently. Coding the offset, which refers either to the two child nodes or to the leaf primitive list, within 29 bits allows for a maximum "node" distance of $2^{29} * 8$ bytes. As $2^{29} * 8 = 2^{32}$, the addressing scheme does not reduce the available address space (on a 32-bit architecture).

Memory access on current CPU architectures works with cache-line granularity. The typical cache-line size is 32, 64 or 128 bytes depending on the underlying CPU architecture. This means that the CPU loads/stores a complete cache-line no matter the actual operand size of the memory access. Having a kd-tree memory alignment of more than $2^3$ ensures that accessing the first child node automatically fetches the second child node. Depending on cache-line size, $8 - 16$ nodes can be stored within a single cache-line. For certain cases it could be beneficial to regroup the kd-tree nodes in small subtrees of cache-line size [Havran01]. This technique might be able to further reduce the number of cache misses during traversal.

```
// efficient 8-byte layout for a kd-tree node

struct KDTreeNode {
  union
  {
    float split_position; // position of axis-aligned split plane
    unsigned int items;   // or number of leaf primitives
  }
  unsigned int dim_offset_flag;
  // the 32 bits of 'dim_offset_flag' are used to encode multiple data
  // bits[0..1]  : encode the split plane dimension
  // bits[2..30] : encode an address offset
  // bit[31]     : encodes whether a node is an inner node or a leaf
};

// macros for extracting node information
#define ISLEAF(n)    (n->dim_offset_flag & (unsigned int)(1<<31))
#define DIMENSION(n) (n->dim_offset_flag & 0x3)
#define OFFSET(n)    (n->dim_offset_flag & 0x7FFFFFFC)
```

*Figure 4.1: kd-tree node layout using only eight bytes. In combination with a proper memory alignment of more than $2^3$ bytes, the kd-tree node layout allows for storing the split dimension within the two lowest bits. The highest bit decides between inner and leaf node, while the remaining 29 bits encode an address offset (substituting a pointer) to either the child nodes or to an index list of primitives.*

## 4.2 Ray Bundle Traversal I

Fast traversal through a kd-tree is essential for achieving high ray tracing performance because the operation is the one most frequently performed. In order to provide a better understanding of the traversal algorithm for bundles, a brief review of the algorithm for single rays is given first.

### 4.2.1 The Ray-Segment Algorithm for Single Rays

The traversal algorithm basically consists of the following operations: Descend from the kd-tree root node in a front-to-back manner (required by early ray termination) by either following the front, the back, or both children until a leaf is reached. In the case of following both children, continue with the front child and push the kd-tree node referring to the back child onto a stack. If a leaf is reached, test all entries in the leaf's primitive reference list (typically, indices referring to a list of scene primitives) for intersection.

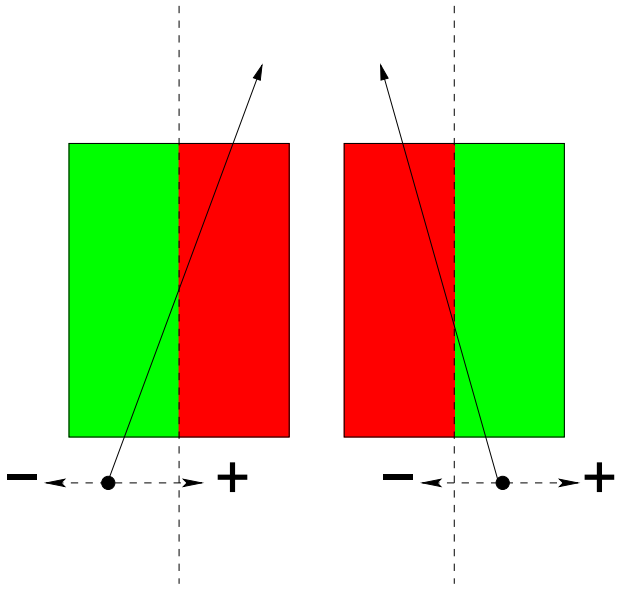



*Figure 4.2: The front "child" (green) and back "child" (red) is determined by the direction of the ray. Therefore, the traversal order can directly be extracted from the direction sign with respect to the dimension of the split plane.*

After testing all primitives of a leaf, the ray is tested for early ray termination. If the ray is not terminated yet and the traversal stack is not empty, a kd-tree node is popped from the stack and the traversal is continued from this node. These steps are performed until the stack is empty or the ray has found a valid intersection. Note that the front respectively back child depend only on the direction of the ray (see Figure 4.2). Moreover, the order can be directly extracted out of the direction sign with respect to the dimension of the split plane. For tracing single rays, this order is fixed during traversal.

The crucial part of the traversal algorithm is to quickly evaluate the decision of whether to follow only the front child, back child, or both children while descending within the kd-tree. The algorithm proposed by Wald et al. [Wald04] operates in one dimension along a ray and is based on the comparison of ray segments. In the following, this algorithm will be called *ray segment algorithm*.

Representing a ray as $R(t) = origin + t * direction$ allows for associating a ray segment $[near, far]$ with parameter $t$. This interval represents the segment where the ray intersects a given cell. The ray segment is first initialized to $[0, \infty)$ and then clipped according to the axis-aligned bounding

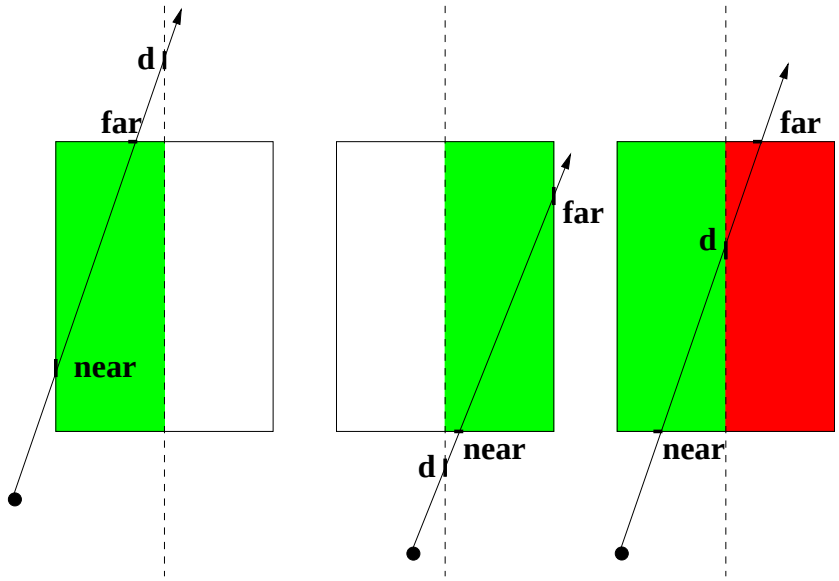



*Figure 4.3: Left: The ray enters only the front child because distance value 'd' is greater than the value 'far'. Center: The ray enters the back child, distance value 'd' being smaller than the value 'near'. Right: The ray enters both children. The traversal is continued with the front child while the back child is pushed onto a stack. For both children, the ray segment has to be updated accordingly.*

box of the scene. This clipping operation initializes $near$ and $far$, which avoids unnecessary traversal operations in the case the origin of a ray lies outside the axis-aligned bounding box of the scene.

As shown in Figure 4.3, testing of the three cases (i.e. following either the front child, back child, or both children) can be realized efficiently by comparing the computed distance $d$ to the splitting plane with the current ray segment $[near, far]$. Three cases are possible:

**$d > far$:** The ray only intersects the cell associated with the front child. The ray segment stays the same.

**$d < near$:** The ray only intersects the cell associated with the back child. The ray segment stays the same.

**$near \leq d \leq far$:** The ray intersects both children. In a first step, the cell in front of the splitting plane must be traversed with [near,d] as the ray segment, and in a second step, the cell behind the splitting plane, with [d,far] as the ray segment.

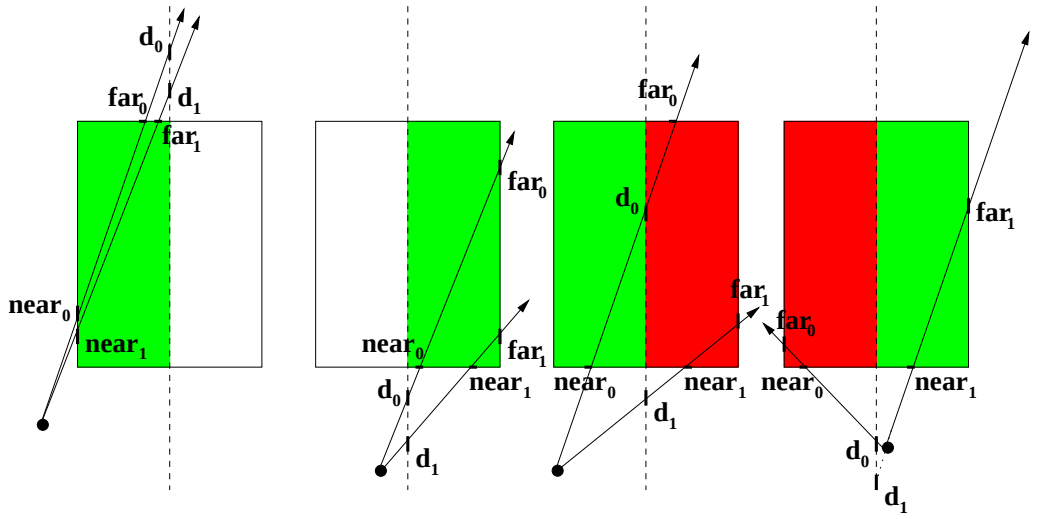



*Figure 4.4: For all ray bundles a common origin is assumed. Left: All rays enter the cell of the front child as all distance values '$d_i$' are greater than the values '$far_i$'. Center left: All rays enter the cell of the back child as all distance values '$d_i$' are lower than the values '$near_i$'. Center right: If only a single ray traverses both children, all rays have to follow. Right: The cells of the front and the back child have to be accessed again, but the ray bundle has different direction signs. Even in this case, a common front-back decision is possible: All rays first access the cell of the child containing the origin.*

One advantage of the ray segment algorithm is that it requires only a short sequence of simple instructions. In particular, only the parametric distance $d$ has to be computed, followed by the two comparisons with the $near$ and $far$ value. Based on these comparisons, one of the three cases has to be determined and the traversal is continued accordingly.

### 4.2.2 The Ray-Segment Algorithm for Ray Bundles

Traversal for bundles of rays relies on the same idea of comparing ray segments as shown for single rays (see Section 4.2.1). Instead of a single interval $[near, far]$, $n$ intervals $[near_i, far_i]$, where $n$ is the number of rays within the bundle, have to be used. Note that in the following it is assumed that all rays within a bundle have a common origin, while the case of arbitrary origins will be discussed later on. This assumption allows for a common traversal order in the event of varying ray direction signs.

Figure 4.4 illustrates that if the segment comparison for all rays is identical ($d_i > far_i$ or $d_i < near_i$), only one child has to be traversed similarly to the single ray case. In all other cases, both children must be traversed. In other words: If any ray indicates a specific child, all rays have to follow. This illustrates why coherence within the ray bundle is essential for achieving high performance, because the more coherent the rays are the more likely it is that all rays make the same decision.

Attention must be given to determining the front and back order when both children have to be traversed. In the case that all rays have the same direction sign (with respect to the split dimension), the front and back child is unambiguously determined (as in the single ray case). If the ray direction signs vary but (as assumed) all rays have a common origin, a non-ambiguous order (see Figure 4.4) is still ensured, because all rays first traverse the child containing the origin.

Computing the appropriate order every time from scratch during traversal can be quite complex. Therefore, it can be beneficial to split the rays into bundles with equal direction signs before the traversal. Moreover, in practice only a few bundles need to be split up. For more simplicity, the case of equal direction signs is assumed for all subsequent code examples.

A similar procedure can be applied if the rays within the bundle do not share a common origin and ray directions differ. In this case, no consistent front and back order can be determined. The best way to ensure a consistent order is to split the rays into bundles containing only rays with equal ray directions ensuring a common traversal decision.

### 4.2.3 Implementation

In the following, an example implementation for the ray segment traversal algorithm for ray bundles is proposed. The implementation is optimized for exploiting SIMD features by using SSE intrinsics. As SSE operates on four floating point values at once, it is beneficial to use ray bundles with a multiple of four rays. The implementation is divided into three parts: Initialization, traversal order determination, and the traversal operation itself.

#### Initialization

The initial clipping against the scene bounding box computes the minimum and the maximum parametric distances to all three plane slabs [Kay86] (a box is defined as the intersection of three slabs). Figure 4.5 shows the SSE implementation for a bundle of four rays. Note that values of *clip_min* and *clip_max* will be exchanged according to the ray directions, before updating

*near4* and *far4*. During traversal, the parametric distance $d$ to the splitting plane is computed as $d = (split - origin) * 1/direction$. By precomputing $1/direction$ a costly division operation is avoided. Note that the inverse computation does not have to be performed at this point, e.g. the inverse of directions for primary rays can be initialized when the rays are generated.

```
sse_t near4 = _mm_setzero_ps();     // -- four ray segments --
sse_t far4 = _mm_set_ps1(Infinity);
SSEVec4 oneOverDirection;           // -- 1.0 / ray4.direction --

// -- clipping in the x dimension --

oneOverDirection4.t[0] = Inverse4(ray4.direction.t[0]);

// compute minimum distance by (box.min.x - origin.x) * 1.0/direction.x
const sse_t clip_min = _mm_mul_ps(_mm_sub_ps(_mm_set_ps1(box.min.x),
                                             ray4.origin.t[0]),
                                  oneOverDirection.t[0]);

// compute maximum distance by (box.max.x - origin.x) * 1.0/direction.x
const sse_t clip_max = _mm_mul_ps(_mm_sub_ps(_mm_set_ps1(box.max.x),
                                             ray4.origin.t[0]),
                                  oneOverDirection4.t[0]);

// update near4 and far4 based on the minimum and maximum distance
const sse_t cmp = _mm_cmpgt_ps(ray4.direction.t[0],_mm_setzero_ps());
near4 = _mm_max_ps(near4,Update4(cmp,clip_min,clip_max));
far4  = _mm_min_ps( far4,Update4(cmp,clip_max,clip_min));
```

*Figure 4.5: Initial segment initialization for a four-ray bundle. The four ray segments are stored within the two SSE variables 'near4' and 'far4'. After initialization, one clipping step against the scene bounding box, here in the first dimension, is applied. Clipping in the other two dimensions is implemented accordingly. The inverse of the directions is stored in the 'oneOverDirection4' structure.*

In order to achieve a common traversal order, it is necessary to split up a ray bundle with different ray directions into bundles with equal ray directions. This case can be quickly detected by comparing the direction signs of all rays within the bundle. The signs for a four-ray bundle can be extracted by a simple *_mm_movemask_ps* instruction. This instruction extracts the sign bits of all register elements, and stores the final 4-bit result within a general purpose register. In the case the four signs are equal (either 0 or $0xf$) with respect to all three dimensions, the bundle does not need to be split. The

splitting operation can actually be seen as a sorting of ray directions into the eight coordinate system octants. Each octant ensures equal ray directions, and can therefore be processed without further splitting.

In the case a ray does not enter the scene bounding box it should not influence any decision in the traversal and intersection. Therefore, such invalid rays have to be identified and deactivated. Fortunately, the clipping step automatically sets the ray segment for invalid rays to the invalid state of $near > far$. Detecting this case by performing an SSE comparison between $near4$ and $far4$ allows for using the bit-mask result as a valid/invalid mask for rays.

```
const unsigned int dir_x = *(unsigned int *)(&ray4.direction.t[0]);
const unsigned int dir_y = *(unsigned int *)(&ray4.direction.t[1]);
const unsigned int dir_z = *(unsigned int *)(&ray4.direction.t[2]);
const unsigned int ray_dir_x = (dir_x & ((unsigned int)1<<31))) >> 28;
const unsigned int ray_dir_y = (dir_y & ((unsigned int)1<<31))) >> 28;
const unsigned int ray_dir_z = (dir_z & ((unsigned int)1<<31))) >> 28;
const unsigned int ray_dir[3][2] = {
  { ray_dir_x,ray_dir_x^((unsigned int)1 << 3) },
  { ray_dir_y,ray_dir_y^((unsigned int)1 << 3) },
  { ray_dir_z,ray_dir_z^((unsigned int)1 << 3) } };
```

*Figure 4.6: Precomputing the traversal order (front and back child) as offsets for a bundle of rays with equal directions. For a given dimension 'd', the front child has an offset of 'ray_dir[d][0]', and the back child of 'ray_dir[d][1]'. In order to efficiently obtain the pointer to the front respectively back child of the current kd-tree node, only the precomputed offset has to be added to the current node address. Note that the size of a kd-tree node is 8 bytes. Therefore, the extracted sign bit must be right-shifted by 28 bits. The offset table contains only the values 8 or 0.*

### Traversal Order

Under the condition of a common traversal order, it is only necessary to decide which of the left or right kd-tree child node is the front respectively back child. By using a small array of two integer offsets per dimension, as shown in Figure 4.6, the offset to the front and back child can be precomputed (based on the direction signs) for the whole bundle. As all rays within the bundle have equal directions signs, the simplification can be made to initialize the offset table with the data of the first ray of the bundle.

The offset table itself allows for speeding up the inner-traversal loop because it avoids recomputing the offset for every traversal step. Computing

the address of the child node now only requires a simple integer addition to the address stored in the parent node. This is however only a small optimization: The offset table and its impact on the inner-most traversal loop can typically increase the total ray tracing performance by up to a few percent.

### Traversal

The core of the actual traversal code basically consists of the decision of whether to follow either the left, right, or both child nodes. As shown in Figure 4.4, the state of all (active) rays must be considered in the decision. Similar to the initial clipping, attention must be paid to rays that enter a cell that they did not intend to enter. These rays must not influence any decision regarding the traversal of the sub-tree that they are forced to enter. Therefore, these rays have to be deactivated.

Ray segments of such invalidated rays are automatically set to the invalid state of $near > far$. Therefore, identifying invalid rays can be efficiently implemented by testing their current $near$ and $far$ values during traversal. The bit mask returned by the SSE comparison can be used (in combination with the initial state provided by the clipping step) to deactivate these rays. Note that the invalidation of rays can only happen when the ray bundle traverses both children. Another way for fast ray invalidation is to track the state (active/inactive) of each ray during traversal. If a ray bundle intends to traverse both children, two masks are created which correspond to the active rays within each sub-voxel. This mask is then logically combined with the current active mask. Therefore, no deactivated ray can become active again. Note that in the case of explicitly handling the active mask, the mask of the back sub-voxel has to be saved on the stack. Experiments have shown that the performance of these two approaches is equivalent.

Figure 4.7 shows the actual traversal code. The inner-most loop and therefore the most time-critical part descends within the kd-tree until a leaf entry is found. For each traversal step, the dimension is extracted first. Based on the dimension, the four distances $d$ to the splitting plane are computed. This computation involves an *_mm_set_ps1* instruction, which is actually a compound of *_mm_set_ss* and *_mm_shuffle_ps*, to copy the split value into the four register elements, an *_mm_sub_ps* instruction for subtracting the ray origins, and finally an *_mm_mul_ps* for a multiplication with the inverse directions. The computation can be interleaved with the computation of the address of the two child nodes and the determination of the front and back child, because the latter only depends on dimension $k$. Interleaving instructions is beneficial to address pipeline latencies. The address computation only involves integer instructions, while computing the distances $d$ only targets

```
unsigned int activeMask = 0xf, termination = 0, index = 0;
union { KDTreeNode *node; unsigned int adr; };
node = scene->root; // root of kd-tree
do { // traverse until a leaf is reached
 while (!ISLEAF(node)) {
   const unsigned int k = DIMENSION(node);
   const sse_t d = _mm_mul_ps(_mm_sub_ps(_mm_set_ps1(node->split),
                                         ray4.origin.t[k]),
                              oneOverDirection4.t[k]);
   adr += OFFSET(node);
   KDTreeNode *back = (KDTreeNode *)(adr + ray_dir[k][1]);
   KDTreeNode *front = (KDTreeNode *)(adr + ray_dir[k][0]);
   node = back;  // only back child traversal ?
   if (!(_mm_movemask_ps(_mm_cmple_ps(near4,d)) & activeMask)) continue;
   node = front; // only front child traversal ?
   if (!(_mm_movemask_ps(_mm_cmpge_ps(far4,d)) & activeMask)) continue;
   stackNode[index] = back; // push values on stack
   stackSSE4[index].near4 = _mm_max_ps(near4,d);
   stackSSE4[index].far4 = far4;
   index++;
   far4 = _mm_min_ps(far4,d); // update far values and the active mask
   activeMask &= _mm_movemask_ps(_mm_cmple_ps(near4,far4));
 }
 if (node->items) {
    // ... intersection code ...
    termination |= activeMask &
                   _mm_movemask_ps(_mm_cmpge_ps(far4,hit4.dist));
    if (termination == 0xf) return; // valid intersections -> terminate
 }
 if ( index == 0 ) return; // stack empty -> terminate
 index--; // pop values from the stack and update active mask
 node = stackNode[index];
 near4 = stackSSE4[index].near4;
 far4 = stackSSE4[index].far4;
 activeMask = ~termination & _mm_movemask_ps(_mm_cmple_ps(near4,far4));
} while(1);
```

*Figure 4.7: SSE traversal code for a bundle of four rays (with equal direction signs). The four lowest bits of 'activeMask' correspond to the current ray state (activated/deactivated), which is based on the current relation between 'near4' and 'far4'. The code consists of three parts: Traversal until a leaf is reached, then perform (possible) primitive intersection tests, and if not all rays are set for early ray termination, starting a new traversal iteration with values taken from the stack. When taking an entry from the stack the 'activeMask' is updated with the mask of the already terminated rays.*

the floating point pipeline. The out-of-order execution unit of the processor is able to execute these instructions mostly in parallel.

The decision whether to traverse the front child, back child, or both children is computed by only considering active rays. The lowest four bits of *activeMask* indicate the current state of all four rays. Applying a logical *and* between *activeMask* and the result of SSE comparison between $d$ and $near4$ respectively $far4$, automatically sets the flag register, removing the need for an additional compare instruction. The first comparison between $d$ and $near4$ tests if no ray requires to access the front child. If this condition is true all rays will go to the back child. The second comparison for testing the front child (no ray needs to access the back child) based on $d$ and $far4$, works accordingly.

If both the front and the back children have to be considered, the active mask is updated based on the current values by applying a logical *and* between $near4$ and $far4$. This is done because a deactivated ray can never become active again, so the number of active rays will only become smaller while descending the tree. Additionally, the back node and the corresponding far value are pushed onto the stack, and the corresponding stack index is increased.

The two branches within the inner-most traversal loop are only taken if all rays go either exclusively to the front or back child. These two cases should be arranged before the case of traversing both children, because the more coherent the rays are the lower the probability for the later case. The probability of the front and back branch depends on the scene and in particular on the view, which is why the branch arrangement does not matter significantly. Statistics have shown that for coherent rays all three cases occur with equal probability. Therefore, the two exclusive branches are taken with an average probability of 2/3 and require fewer instructions than the both child case.

In terms of cache misses, the critical memory access occurs when loading the data of a kd-tree node from memory. Reducing the cache misses by inserting prefetch instructions fails because the traversal code is too small to completely hide the memory latency. On current CPU architectures, roughly 200 cycles are required to hide the latency of main memory accesses. On the other hand, the CPU cache hierarchy ensures a reasonable cache hit rate because of the locality of kd-tree node accesses.

When reaching a leaf node an empty test is applied first. As advanced kd-tree construction tends to cut off empty space as much as possible, many empty leaves will be created. In the case that the current leaves is not empty, intersection tests, e.g. for triangles (see Chapter 5) are performed based on a primitive index list.

After performing all intersection tests, all rays are checked for early ray termination. Note that only those rays are marked as terminated whose intersection distance is smaller than the current *far* value and the corresponding active masks are set. If the distance is higher than the current *far* value, a potential new intersection could exist between the current *far* and the intersection distance.

At the end, if the stack is not empty, the corresponding values are taken from the stack and traversal is restarted. Before restarting traversal, the active mask is reset based on the updated *near* and *far* values, and on any newly terminated rays.

The code in Figure 4.7 stores the active mask as an integer bit mask. On some CPU architectures, it is more beneficial to hold the active within SSE registers, and apply an *_mm_and_ps* before transferring the result of the comparison using the *_mm_movemask_ps* instruction. The latter also suffers from a very high latency of 7 or more cycles (on current architectures). The high latency results from the fact that the asynchronously running integer and floating point pipelines have to be synchronized.

Further critical performance hot spots are the two branches because they are purely data dependent. Therefore, the processor's branch prediction unit cannot really reduce the amount of mis-predicted branches. Besides the impact of branch mis-prediction, the traversal code for four rays has a lot of data dependencies, making parallel out-of-order execution difficult. For larger bundles, some operations like the computation of the split plane distances can be done in parallel across all four-ray bundles. Nevertheless, branching according to the three traversal cases requires the comparison results of all four-ray bundles. This introduces implicit synchronization, limiting the code regions for potential parallel execution.

### Extension: Removing Branches

The traversal loop can be implemented without branches (see Figure 4.8) because a mis-predicted branch causes a pipeline stall up to the length of the processor's execution pipeline.

Current CPU architectures allow for using conditional move instructions to prevent branching. These instructions move a value to a register only if a specific condition holds true. Current C/C++ compilers [Intel02a, GNU] tend to use such conditional move instructions through the *?:* operator, if both operands are constant values. The code shown in Figure 4.8 applies this operator for setting the traversal side. Note that the conditional *?:* operator represents only a hint for the compiler to use a conditional move instruction, not a must. Depending on the compiler, it is helpful to manually modify

```
sse_t far4_update[2];
union { KDTreeNode *node; unsigned int adr; };

while (!ISLEAF(node))
 {
  const unsigned int k = DIMENSION(node);
  const sse_t d = _mm_mul_ps(_mm_sub_ps(_mm_set_ps1(node->split),
                  ray4.origin.t[k]), oneOverDirection4.t[k]);
  adr += OFFSET(node);
  far4_update[0] = _mm_min_ps(far4,d);
  far4_update[1] = far4;

  const unsigned int d_near = _mm_movemask_ps(_mm_cmple_ps(near4,d));
  const unsigned int d_near_active = (!(d_near & activeMask)) ? 1 : 0;

  stackNode[index] = (KDTreeNode *)(adr + ray_dir[k][1]);
  node = (KDTreeNode *)(adr + ray_dir[k][d_near_active]);

  const unsigned int d_far = _mm_movemask_ps(_mm_cmpge_ps(far4,d));
  const unsigned int d_far_active  = (!(d_far & activeMask)) ? 1 : 0;

  stackSSE4[index].near4 = _mm_max_ps(near4,d);
  stackSSE4[index].far4 = far4;

  const unsigned int both_sides = d_near_active | d_far_active;
  index += 1 ^ both_sides;
  far4 = far4_update[both_sides];
  activeMask &= _mm_movemask_ps(_mm_cmple_ps(near4,far4));
 }
```

*Figure 4.8: The inner-most traversal loop implemented without branches through applying conditional move instructions (using the ?: operator). Branchless code avoids the penalty of branch mis-predictions. As the complete code sequence is executed regardless of the determined traversal case, more instructions have to be executed. The break-even point, where the branchless version becomes faster depends on the probability distribution of the three traversal cases. Note that the 'far4' value must only be updated with the minimum of 'far4' and 'd' if both children need to be traversed. This code uses a table look-up to properly update the 'far4' value.*

the code in order to achieve the desired goal, e.g. by applying explicit *const* typecasts or code rearrangements.

The conditional move instructions are used to compute a traversal flag that is active in the event that both children have to be traversed. Regardless of the traversal case, the back child is always saved on the traversal stack,

```
#define FOR_ALL_BUNDLES for (unsigned int i=0;i<BUNDLES;i++)
while (!ISLEAF(node))
 {
  const sse_t node_split = _mm_set_ps1(node->split);
  const unsigned int k = DIMENSION(node);
  adr += OFFSET(node);
  KDTreeNode *front = (KDTreeNode *)(adr + ray_dir[k][0]);
  KDTreeNode *back = (KDTreeNode *)(adr + ray_dir[k][1]);
  unsigned int d_near = 0, d_far = 0;
  sse_t d[BUNDLES];
  FOR_ALL_BUNDLES
   {
    d[i]    = _mm_mul_ps(_mm_sub_ps(node_split,ray4[i].origin.t[k]),
                         oneOverDirection4[i].t[k]);
    d_near |= _mm_movemask_ps(_mm_and_ps(_mm_cmple_ps(near4[i],d[i]),
                                         activeMask[i]));
    d_far  |= _mm_movemask_ps(_mm_and_ps(_mm_cmpge_ps(far4[i],d[i]),
                                         activeMask[i]));
   };
  node = back;
  if (d_near == 0) { continue; } // traverse the back child
  node = front;
  if (d_far == 0) { continue; } // traverse the front child
  // traverse both children => push values on the stack
  FOR_ALL_BUNDLES {
    stackISSE[stackIndex].far4[i]  = far4[i];
    stackISSE[stackIndex].near4[i] = _mm_max_ps(near4[i],d[i]);
    far4[i] = _mm_min_ps(far4[i],d[i]);
    activeMask[i] = _mm_and_ps(activeMask[i],
                               _mm_cmple_ps(near4[i],far4[i]));
  }
  stackNode[stackIndex] = back;
  stackIndex++;
}
```

*Figure 4.9: The inner-most traversal loop for an arbitrary number of four-ray bundles. The implementation relies on the same algorithm as shown in Figure 4.7. Note that for each four-ray bundle, the 'activeMask' is stored as SSE data type.*

but the stack pointer is only updated if the traversal flag is active. A similar mechanism is applied for updating the *far4* value: Both the original and a modified *far4* value are stored into a table, and the *far4* is set by a table look-up based on the traversal flag. Updating the *far4* could be implemented by a sequence of conditional move instructions, but a table look-up is usually faster.

As the branchless version executes more instructions as the version with branches, the break-even point where the branchless version becomes faster depends on the probability distribution of the three traversal cases. The more likely the both-side case is, the more beneficial the branchless version will be. Therefore, ray bundles with a low coherence are more suitable for the branchless version.

### Extension: Tracing Multiple Ray Bundles

Extending the code to trace more than four rays in parallel is straightforward. As SSE operates on four elements, it is beneficial to handle multiples of four-ray bundles, e.g. four four-ray bundles corresponding to 16 rays. Note that 16 rays require four *near4* and four *far4* values. Unfortunately, the eight SSE registers (on standard 32-bit Intel or AMD architectures) do not allow for holding all *near4* and *far4* values within registers during traversal. This introduces additional memory accesses (temporary values must be stored to memory). An alternative implementation could store the *activeMask* as an integer value, instead of using the *sse_t* data type. Which way is actually faster, largely depends on the actual architecture used.

Figure 4.9 shows the traversal loop for an arbitrary number of four-ray bundles. Note that computing distances $d_i$, and the corresponding comparisons to $near_i$ and $far_i$ for a bundle $i$ is independent of other bundles, allowing for an increased instruction level parallelism.

Moreover, the decision to follow only one side comes up roughly in the middle of the traversal code. This means, the more likely the one-side case happens the more often half of the traversal code is skipped. One could easily implement the traversal code for multiple bundles without branches, following the same approach as shown in Figure 4.8. Whether this approach is beneficial or not, largely depends on the probability that both children must be traversed as well as the penalty cost for a mis-predicted branch.

### Extension: Tracing Multiple Ray Bundles With Masking

Apart from the given code examples, many implementation variants are possible. As mentioned in the beginning of this section, storing the active mask as integer directly on the stack will avoid updating the active mask based on the current near and far values. Moreover, the complete front/back child determination can be recoded in a left/right child decision using a sequence of Boolean expressions (see Figure 4.10).

As the fact that the front or back child only depends on the ray directions, they can be evaluated on the fly. Therefore, all ray directions (with respect to

```
#define FOR_ALL_BUNDLES for (unsigned int i=0;i<BUNDLES;i++)
unsigned int activeMask = 0xffff;
while (!ISLEAF(node)) {
    const sse_t node_split = _mm_set_ps1(node->split);
    const unsigned int k = DIMENSION(node);
    adr += OFFSET(node);
    unsigned int dgef = 0; unsigned int dlen = 0;
    sse_t d[BUNDLES];
    FOR_ALL_BUNDLES {
      d[i]  = _mm_mul_ps(_mm_sub_ps(node_split,ray4[i].origin.t[k]),
                         oneOverDirection4[i].t[k]);
      dlen |= _mm_movemask_ps(_mm_cmpgt_ps(near4[i],d[i])) << (4*i);
      dgef |= _mm_movemask_ps(_mm_cmpgt_ps(d[i],far4[i]))  << (4*i);
    };
    const unsigned int bactive = ((dgef^0xffff) &
                                  (dlen^0xffff)) & activeMask;
    dgef &= activeMask; dlen &= activeMask;
    // active rays that require to traverse the right child
    const unsigned int ractive = (dgef & dir[k]) |
                                 (dlen & (dir[k]^0xffff)) | bactive;
    // active rays that require to traverse the left child
    const unsigned int lactive = (dgef & (dir[k]^0xffff)) |
                                 (dlen &  dir[k]) | bactive;
    if (lactive == 0) { adr += 8; continue; } // traverse the right child
    if (ractive == 0) { adr += 0; continue; } // traverse the left child
    FOR_ALL_BUNDLES {
      stackISSE[stackIndex].far4[i]  = far4[i];
      stackISSE[stackIndex].near4[i] = _mm_max_ps(near4[i],d[i]);
      far4[i] = _mm_min_ps(far4[i],d[i]);
    }
    const unsigned int side = (bactive & dir[k]) != 0;
    node += side;
    stackNode[stackIndex] = (KDTreeNode*)adr^8;
    stackNode[stackIndex] = side ? lactive : ractive;
    activeMask &= side ? ractive : lactive;
    stackIndex++;
  }
```

*Figure 4.10: The inner-most traversal loop for up to eight four-ray bundles (all rays must have the same direction sign). Instead of holding the ray active mask within multiple SSE registers, the mask is stored as a bit mask within a single integer register (requiring more '_mm_movemask_ps' instructions). The decision of whether to follow the front or back child is modified into a decision of whether to follow the left child (offset 0) or the right child (offset 8).*

the three dimensions) are extracted and stored as bit-masks into three integer variables. Depending on the split dimension, one of these three direction masks is applied to determine the front and back child, respectively.

Compared to the code example of Figure 4.9, the deactivation of invalid rays is not performed by SSE instructions but integer instructions. Depending on the architectures, this modification can be beneficial but in general the two implementations provide similar performance.

#### Future Modifications

If a future version of the SSE instruction set includes a multiply-add instruction, the dependency chain for computing the distances to the splitting plane can be reduced. Instead of performing $(split - origin) * 1/direction$, one could rearrange this code sequence to $split * 1/direction - origin * 1/direction$, while the $-origin * 1/direction$ could be precomputed. The rearrangement allows for directly applying a multiply-add instruction of the form $a * b + c$.

Note that there is no unique fastest code sequence. Depending on the architecture, in particular on the throughput and latency of the instructions, one has to choose an appropriate implementation. Moreover, some optimization techniques are beneficial for a certain architecture, e.g. the Pentium-IV, but are disadvantageous for other architectures such as the Opteron.

## 4.3 Ray Bundle Traversal II

Reshetov et al. [Reshetov05] proposed an alternative traversal algorithm which does not consider the decision of each ray in order to determine the behavior of the whole bundle. Instead, the algorithm efficiently culls parts of a kd-tree which are known not to be intersected by the frustum spawned by the rays within the bundle. The underlying culling technique is called *inverse frustum culling*.

### 4.3.1 The Inverse Frustum Culling Algorithm

*Inverse frustum culling* allows for efficiently culling the axis-aligned bounding boxes (AABBs) associated with kd-tree cells by using the faces of a given AABB as separation planes. If one of these planes separates the ray bundle from the AABB, the corresponding cell can be efficiently culled.

Figure 4.11 illustrates the approach by using a 2D example. A given cell associated to a kd-tree node is split into two sub-cells (associated to the two

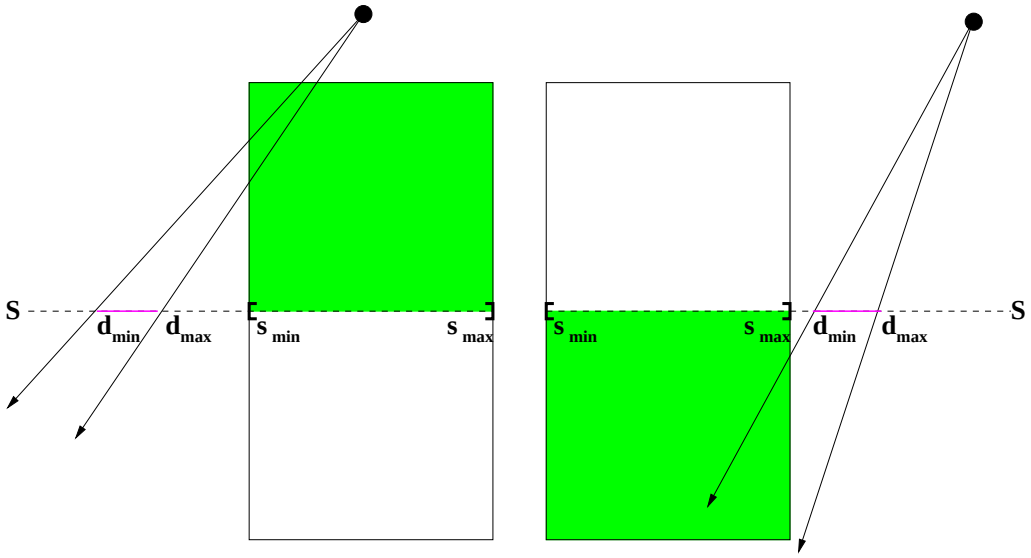



*Figure 4.11: Traversal for ray bundles by comparing only extremal values: If the extremal intersection value '$d_{max}$' ('$d_{min}$') of all intersections with splitting plane 'S' is less (greater) than '$s_{min}$' ('$s_{max}$') (the bounds of 'S' with respect to the current cell), only the front (back) cell, colored green (white), has to be considered further, while the back (front) is completely culled. The non-culled cell is marked green. If none of the two cases is valid, both cells have to be considered. Note that the culling test has to be applied in two dimensions, i.e. given split dimension 'x', the two culling tests have to be applied to the other two dimension 'y' and 'z'.*

child nodes) by split plane $S$. In a first step, the algorithm computes the intersection values $d_i$ for all rays within the bundle with splitting plane $S$. On the assumption that the bundle intersects the given cell, the intersected sub-cell(s) has/have to be determined. Determining the sub-cell can be efficiently realized by only comparing the extremal values of all ray-splitting-plane intersections $d_i$ with the bounded interval $[s_{min}, s_{max}]$ of splitting plane $S$.

Knowing the extremal values of ray directions in advance allows for handling large bundles of rays without considering all rays during traversal. Assuming a standard perspective camera for generating primary rays, these extremal values correspond to the corner rays of a small $N \times N$ pixel block. The definition of all rays within the bundle must only be known when entering a non-empty leaf, especially for performing the ray-primitive intersection tests.

Depending on the common direction of rays within the bundle, only two cases have to be considered:

**$d_{max} < s_{min}$:** All intersection points $d_i$ are lower than $s_{min}$. All rays will only intersect the cell associated to the front child.

**$d_{min} > s_{max}$:** All intersection points $d_i$ are greater than $s_{max}$. All rays will only intersect the cell associated to the back child.

In all other cases both children have to be traversed.

Applying the algorithm to 3D requires the comparison to be made in two dimensions: For example, given a split plane parallel to the $yz$-plane the intervals in $y$ and $z$ direction have to be used. Furthermore, not only the front and back child but also the roles of $d_{min}$ and $d_{max}$ depend on the common ray directions within the bundle.

### 4.3.2 Implementation

In contrast to the algorithm in Section 4.2, the traversal does not rely on handling ray segments but on continuously updating the axis-aligned bounding box referring to the current kd-tree voxel. Representing the axis-aligned bounding box by the two extremal points, as shown in Figure 3.10, allows for efficiently updating the current box. The extremal component of the box which has to be updated depends on the common ray directions.

#### Initialization

As pointed out, certain parts of the algorithm depend on the common ray directions. The eight possible cases, corresponding to the eight octants, could be implemented separately. The appropriate case only needs to be determined once and in particular before the actual traversal starts. In the following, a different implementation is used. The determination of the traversal case is done in the inner-most traversal code but using data-dependent reads. Therefore, using look-up tables, e.g. for storing integer offsets to extremal points with the box structure (see Figure 4.12) allows for avoiding costly branches within the traversal code.

#### Traversal

The actual traversal code for four rays is shown in Figure 4.13. In a first step, the four distances $t$ to the current split plane are computed. Depending on the dimension of the split, the other two dimensions $dim0$ and $dim1$ are

```
const unsigned int signs[3] = {
    _mm_movemask_ps(ray4.direction.t[0]),
    _mm_movemask_ps(ray4.direction.t[1]),
    _mm_movemask_ps(ray4.direction.t[2])
};
const unsigned int nearOffset[3] = {
    (signs[0] == 0xf) ? 0+0 : 4+0,  // min,max
    (signs[1] == 0xf) ? 0+1 : 4+1,  // min,max
    (signs[2] == 0xf) ? 0+2 : 4+2,  // min,max
};
const unsigned int farOffset[3] = {
    (signs[0] == 0xf) ? 4+0 : 0+0,  // max,min
    (signs[1] == 0xf) ? 4+1 : 0+1,  // max,min
    (signs[2] == 0xf) ? 4+2 : 0+2,  // max,min
};
const unsigned int dimTable[3][2] = {
 {1,2}, // y,z
 {0,2}, // x,z
 {0,1}  // x,y
};
```

*Figure 4.12: The 'signs' table stores the sign bits per dimension for a bundle of four rays. The 'nearOffset' and 'farOffset' tables refer to components of extremal points within the box structure. These tables are efficiently used within the traversal code to avoid costly branches. The last table is used for determining the two dimensions that are different from the current split dimension.*

determined. Computing the intersections points $Y$ and $Z$ with the splitting plane using the two dimensions, allows for an efficient comparison against the bounding box intervals. The minimum respectively maximum component of these box intervals is determined by applying a look-up in the *nearOffset* and *farOffset* table. The decision of whether the front child, back child, or both children are traversed requires only a comparison between the four intersection points and the minimum and maximum box components.

Instead of maintaining a stack of ray segments, a stack of axis-aligned bounding boxes (in addition to the kd-tree node stack) is used. When both children have to be traversed, the current bounding box and the corresponding kd-tree node are pushed onto the respective stack.

Note that early ray termination requires a maximum ray distance with respect to the current leaf. The distances can either be computed during traversal, which is rather costly, or be computed by a sequence of clipping steps when entering a non-empty leaf (see Figure 4.5).

```
while (!ISLEAF(node))
  {
    const float split = node->split;
    const sse_t node_split = _mm_set_ps1(split);
    const unsigned int k = DIMENSION(node);
    (unsigned int)node += OFFSET(node);
    KDTreeNode *front = (KDTreeNode *)(adr + ray_dir[k][0]);
    KDTreeNode *back = (KDTreeNode *)(adr + ray_dir[k][1]);
    const sse_t t = _mm_mul_ps(_mm_sub_ps(node_split,ray4.origin.t[k]),
                               oneOverDirection4.t[k]);
    const unsigned int dim0 = dimTable[k][0];
    const unsigned int dim1 = dimTable[k][1];

    const sse_t Y = _mm_add_ps(ray4.origin.t[dim0],
                               _mm_mul_ps(t,ray4.direction.t[dim0]));
    const sse_t Z = _mm_add_ps(ray4.origin.t[dim1],
                               _mm_mul_ps(t,ray4.direction.t[dim1]));
    const sse_t b_minY = _mm_set_ps1(((float*)&cbox)[nearOffset[dim0]]);
    const sse_t b_minZ = _mm_set_ps1(((float*)&cbox)[nearOffset[dim1]]);
    if ( _mm_movemask_ps(_mm_cmplt_ps(Y,b_minY)) == signs[dim0] ||
         _mm_movemask_ps(_mm_cmplt_ps(Z,b_minZ)) == signs[dim1] )
      {
        ((float*)&cbox)[nearOffset[k]] = split; node = front; continue;
      }
    const sse_t b_maxY = _mm_set_ps1(((float*)&cbox)[farOffset[dim0]]);
    const sse_t b_maxZ = _mm_set_ps1(((float*)&cbox)[farOffset[dim1]]);
    if ( _mm_movemask_ps(_mm_cmpgt_ps(Y,b_maxY)) == signs[dim0] ||
         _mm_movemask_ps(_mm_cmpgt_ps(Z,b_maxZ)) == signs[dim1] ||
         _mm_movemask_ps(t) == 0xf)
      {
        ((float*)&cbox)[farOffset[k]] = split; node = back; continue;
      }
    stackBox[stackIndex] = cbox;
    stackNode[stackIndex] = back;
    ((float*)&cbox)[nearOffset[k]] = split;
    ((float*)&stackBox[stackIndex])[farOffset[k]] = split;
    node = front;
    stackIndex++;
  }
```

*Figure 4.13: Traversal code using extremal properties of the ray bundle. The decision of whether the front child, back child, or both children are traversed is based on the decision of whether all intersection points between the rays and the splitting plane lie outside the rectangular plane region bounded by the current kd-tree voxel.*

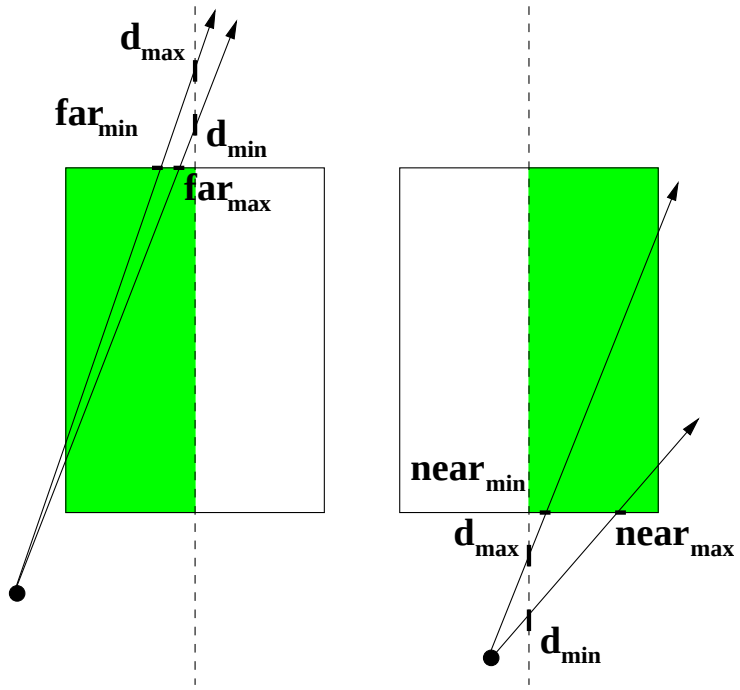



*Figure 4.14: Alternative traversal using the ray segment algorithm for tracing ray bundles (see Section 4.2.2) by comparing extremal values. Left: If the minimum distance '$d_{min}$' to the current split plane is greater than the maximum distance to the current AABB '$far_{max}$', only the front child (marked green) must be traversed. Right: If the maximum distance to the current split plane '$d_{max}$' is less than the minimum distance to the current AABB '$near_{min}$', only the back child (marked green) must be traversed. In all other cases, both children must be considered. In contrast to the standard ray segment algorithm for bundles, only a single ray segment $[near_{min}, far_{max}]$ is used (and updated) during traversal.*

### Alternative Traversal

Compared to the previous algorithm, an alternative traversal (see Figure 4.14) relies on computing the minimum and maximum distance to the AABB corresponding to the current kd-tree node. If the maximum (minimum) of all distances to the current split plane is less (greater) than the current minimum (maximum), all rays will exit the AABB after (before) intersecting the split plane and will therefore only visit the back (front) child.

The code for the alternative traversal algorithm is shown in Figure 4.15. The main advantage of the implementation is the lower code complexity as

compared to the previous implementations for tracing ray bundles. Since only a single ray segment based on extremal values is used, the algorithm will introduce unnecessary traversal steps in the case of low ray coherence within the bundle.

Note that for further optimization, the horizontal operation within the code shown in Figure 4.15 can be removed by using a precomputed table of minimum and maximum values of $1/ray.direction$ for determining $dMin$ and $dMax$.

```
// float near: the current minimum of all distances
// float far: the current maximum of all distances
while (!ISLEAF(node))
  {
    const float split = node->split;
    const sse_t node_split = _mm_set_ps1(split);
    const unsigned int k = DIMENSION(node);
    adr += OFFSET(node);
    KDTreeNode *front = (KDTreeNode *)(adr + ray_dir[k][0]);
    KDTreeNode *back = (KDTreeNode *)(adr + ray_dir[k][1]);
    const sse_t d = _mm_mul_ps(_mm_sub_ps(node_split,ray4.origin.t[k]),
                               oneOverDirection4.t[k]);
    const float dMin =  _mm_cvtss_f32(sseHorizontalMin(d));
    const float dMax =  _mm_cvtss_f32(sseHorizontalMax(d));
    node = back;
    if ( dMax < near) { continue; }
    node = front;
    if ( dMin > far) {  continue; }
    stack[stackIndex] = back;
    stack[stackIndex].near = MAX(dMin,near);
    stack[stackIndex].far =  far;
    stackIndex++;
    far = MIN(dMax,far);
  }
```

*Figure 4.15: Alternative traversal variant for handling ray bundles by extremal values. The code is based on comparing the extremal distances 'dMin' and 'dMax' (to a splitting plane) with a single ray segment* $[near, far]$.

### 4.3.3 kd-Tree Entry Point Search

Reshetov et al. [Reshetov05] proposed to use one of the two previous extremal traversal algorithms to find better entry points within the kd-tree. Having deep entry points within the kd-tree allows for minimizing the required number of traversal steps per bundle.

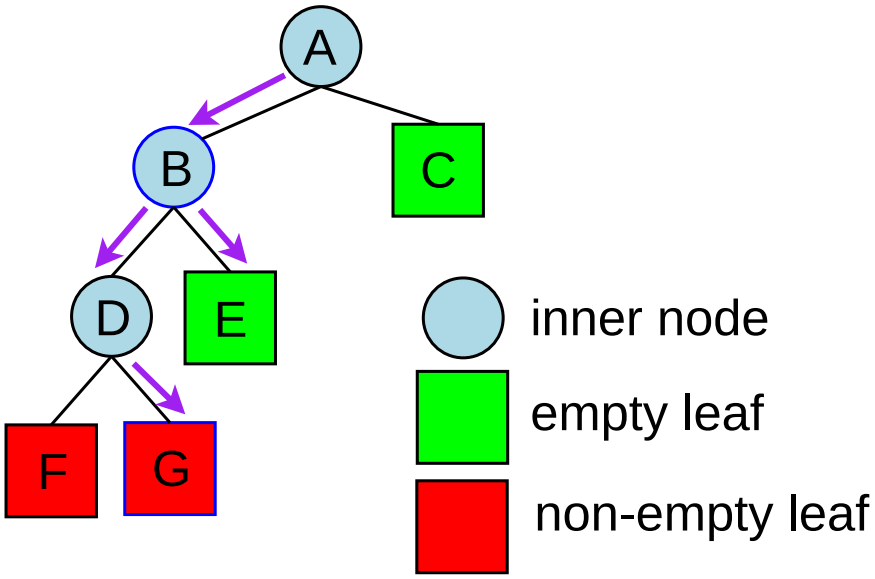



*Figure 4.16: Simple example of finding a kd-tree entry point for a bundle of rays using extremal traversal: Starting at the root node 'A', only the left child 'B' has to be traversed. At 'B', both children have to be considered. Therefore, 'B' is pushed onto the bifurcation stack, and the traversal continues with the front child 'D' of 'B'. At 'D', only the right child 'G' has to be traversed. Node that 'G' is a non-empty leaf (marked red), so it becomes the first kd-tree entry point candidate and the bifurcation stack is frozen. As the bifurcation stack is non-empty (it contains 'B'), the entry point algorithm takes a node from the bifurcation stack and continues with its back child (the front child has already been traversed). The back child 'E' is an empty leaf (marked green), so the current kd-tree entry point candidate is not updated. As 'E' is a leaf and the bifurcation stack is empty, the current candidate 'G' is returned as the kd-tree entry point. In the event that the entry point is already a leaf, which holds true for 'G', only primitive intersection tests need to be performed (skipping ray bundle traversal).*

The idea behind the kd-tree entry point search is to start with a frustum that defines the traversal behavior of a bundle of contained rays. For primary rays generated by a standard perspective camera, the corner rays of a small $N \times N$ pixel tile will define an appropriate frustum. The ray frustum is traversed through the kd-tree until a non-empty leaf is found (see Figure 4.16), or more precisely, until the axis-aligned bounding box of all primitives within the non-empty leaf is intersected by the frustum. During traversal, all *bifurcation* kd-tree nodes (all nodes where both children have to be considered)

are pushed onto a stack. This stack (including the first non-empty leaf) contains all potential kd-tree entry point candidates. The algorithm continues by taking a potential candidate from the stack and traversing the corresponding back child (the front child has already been traversed). If for an entry candidate a non-empty leaf is found, the corresponding bifurcation node becomes the current entry point candidate. The traversal and candidate updating continues until the stack is empty, all rays within the frustum intersect the primitives of a non-empty leaf, or a leaf is empty but is marked as inside some "watertight" object ("empty occluders") [Reshetov05]. In all cases, the current candidate is returned as an entry point. Since not all rays within the frustum have to be defined at the point where the frustum is tested for termination, the intersection test must rely on the properties of the frustum, i.e. if the frustum misses a primitive, all rays within the frustum will miss the primitive too.

In addition to returning the kd-tree entry point, the algorithm must also return the corresponding kd-tree cell as an axis-aligned bounding box. Subsequent traversal algorithms are initialized by this AABB. An inaccurate AABB, as for example the scene bounding box, will cause unnecessary traversal overhead. Note that in the case of ray segment-based traversal, the corresponding ray clipping step must be performed with the AABB returned by the kd-tree entry point algorithm.

With respect to a given ray frustum, the kd-tree entry point algorithm returns either the first bifurcation node, or in the event the bifurcation node is empty, the leaf where all rays have ended. The complete sub-tree above the entry point can be excluded for the entire set of rays which are bounded by the frustum. The more coherent the rays are, the deeper the entry point is located within the kd-tree. This allows for efficiently reducing the number of traversal steps required for the remaining set of rays. In order to achieve good entry points, Reshetov [Reshetov05] suggested to start (for primary rays) with boundary rays for a fairly large set of rays to continuously create smaller sets, and to find respective entry points for them. Section 4.4 presents the corresponding results when the entry point algorithm is used for primary rays.

Compared to a standard traversal of ray bundles, the entry point algorithm does not perform any intersection test because usually not all rays have been defined at this point. For the actual implementation, both traversal variants shown in Figure 4.13 and Figure 4.15 can be used (with the modification of storing the parent kd-tree node on the stack, instead of the node referring to the back child). Note that the latter procedure results in a more inaccurate traversal because of using and continuously updating a single ray segment of extremal values. The inaccurate traversal will typically

result in an increased number of traversal and intersection steps. However, these additional intersection tests can be reduced by performing an additional clipping step against the axis-aligned bounding box spawned by the primitives within a non-empty leaf [Reshetov05].

On the other hand, the implementation of the second traversal variant requires significantly fewer instructions compared to the first traversal variant. For either traversal variant, no masking of inactive invalid rays can be applied (again, not all rays have to be defined at this point). If these traversal algorithms are applied for standard traversal, the rays have to be coherent or significant traversal overhead will be caused (depending on the kd-tree quality). In the case these algorithms are applied for rather incoherent rays, Reshetov [Reshetov05] proposes to use extremal traversal in the upper hierarchy while changing to the standard traversal (with masking) in the deeper parts of the kd-tree (see Section 4.4).

## 4.4 Experiments and Results

In order to illustrate the benefits of tracing ray bundles, three triangular example scenes with varying complexity will be discussed in the following (see Figure 4.17). All models are rendered at a resolution of $1024 \times 1024$, casting only primary rays.

*Figure 4.17: A small set of triangular example scenes for illustrating traversal statistics in the context of ray bundles. The scene complexity varies from left to right: 280K (Conference), 680K (VW Beetle), 2.16M (Soda Hall) triangles. All test scenes are rendered at a resolution of* $1024 \times 1024$*, using only primary rays.*

### 4.4.1 Ray Bundle Sizes

Table 4.1 shows traversal and intersection statistics for different ray bundle sizes. These statistics have been generated for the VW Beetle scene using the standard ray-segment algorithm shown in Figure 4.9.

| Rays/Bundle | 1 | $2 \times 2$ | $4 \times 4$ | $8 \times 8$ | $16 \times 16$ | $32 \times 32$ |
|---|---|---|---|---|---|---|
| T Steps/Bundle | 29.79 | 32.25 | 36.81 | 56.33 | 104.32 | 261.23 |
| T Steps/Ray | 29.79 | 8.06 | 2.3 | 0.88 | 0.41 | 0.26 |
| Only Front Side % | 36.39 | 35.80 | 34.66 | 31.97 | 27.34 | 22.68 |
| Only Back Side % | 39.08 | 38.22 | 36.01 | 31.65 | 25.23 | 19.22 |
| Both Sides % | 24.53 | 25.97 | 29.33 | 36.39 | 47.43 | 58.10 |
| I Steps/Bundle | 2.73 | 3.59 | 5.61 | 11.68 | 29.34 | 88.50 |

*Table 4.1: Traversal and intersection statistics (for primary rays) in relation to different ray bundle sizes. Even though the statistics are generated from the VW Beetle model (see Figure 4.17), the two other scenes show a similar behavior. The average traversal steps per ray (T Steps/Ray) can be significantly reduced by using ray bundles (T Steps/Bundle), while the number of required intersection steps (I Steps/Bundle) increases for larger bundles. Given the higher intersection than traversal costs, a moderate ray bundle size of* $4 \times 4$ *rays offers the best compromise between traversal step reduction and intersection overhead.*

While the total number of traversal steps for a complete frame can be efficiently reduced by using larger ray bundles, the total performance advantage decreases due to the traversal and intersection overhead caused by non-coherent rays. Because of the overhead, the average number of intersections per bundle (*I Steps/Bundle*) increases significantly with a larger bundle size. The impact of traversal overhead for larger bundles can be clearly derived from the relative percentage of the three traversal cases (front child, back child, or both children). Using small bundles almost 75% of the cases require only the traversal of either the front or the back child. This amount decreases to 41% for $32 \times 32$ bundles.

Because of the generally higher costs for intersection than for traversal, a compromise between traversal reduction and intersection overhead has to be found. The chosen bundle size depends on the cost for a traversal step, the cost for an intersection step, and the quality of the kd-tree itself. For the algorithms presented here, and the target resolution of $1024 \times 1024$, a bundle size of $4 \times 4$ rays was found to be most suitable.

If the bounding box of all primitives within a leaf is available, an addi-

tional clipping step between the bounding box and sub-sets of the ray bundle can be applied. Assuming for example a bundle size of $8 \times 8$, additional clipping tests against the four $4 \times 4$ bundles could be used to avoid unnecessary intersection computations. This could further reduce the number of intersection tests per bundle while introducing additional computations per non-empty leaf.

### 4.4.2 Applying kd-Tree Entry Point Search

Finding better kd-tree entry points using the algorithms from Figure 4.13 and Figure 4.15, allows for further reducing the number of traversal steps per bundle. In order to simplify matters, the termination criteria of the kd-tree entry search have been reduced to a simple leaf emptiness test, ignoring advanced criteria such as "empty occluders".

For primary rays, the ray frustum is done by splitting the screen into a starting set of $64 \times 64$ pixel tiles. After looking for a kd-tree entry point by performing an entry search using the ray frustum of a $64 \times 64$ tile (assuming a standard perspective camera for generating primary rays, these extremal values correspond to the corner rays), the tile is split again into a set of 16 sub-tiles with a size of $16 \times 16$ pixels. For each of these $16 \times 16$ pixel tiles, the entry point search is continued by using the entry point of the preceding $64 \times 64$ tile. At this point, the entry search is finished, and the standard traversal is invoked. Therefore, each $16 \times 16$ sub-tile is split once more into 16 tiles of $4 \times 4$ pixel, and a standard traversal algorithm starts with the entry point returned by the entry search of the $16 \times 16$ tile.

For the three example scenes, the average number of traversal steps per $4 \times 4$ bundle can be reduced by up to 48% (see Table 4.2). By fine-tuning the heuristics used for the kd-tree construction for individual scenes (e.g. by varying the threshold for cutting off empty space), the reduction can be further increased to over 50% percent. Note that the kd-tree heuristics (e.g. maximum depth criteria or empty space thresholds) for one particular scene are likely to be sub-optimal for others. This should make it clear that the results achieved by applying kd-tree entry points algorithms depend largely on the quality of the constructed kd-tree itself. Moreover, the integration of more sophisticated termination criteria for the kd-tree entry point search, e.g. through inserting empty occluders [Reshetov05] should provide even better entry points, and therefore a larger reduction of the required traversal steps.

Note that the 48% of saved traversel steps are only achieved for scenes with a rather high geometric occlusion, as for example the conference scene. Figure 4.18 visualizes kd-tree entry points for this scene. Almost all primary

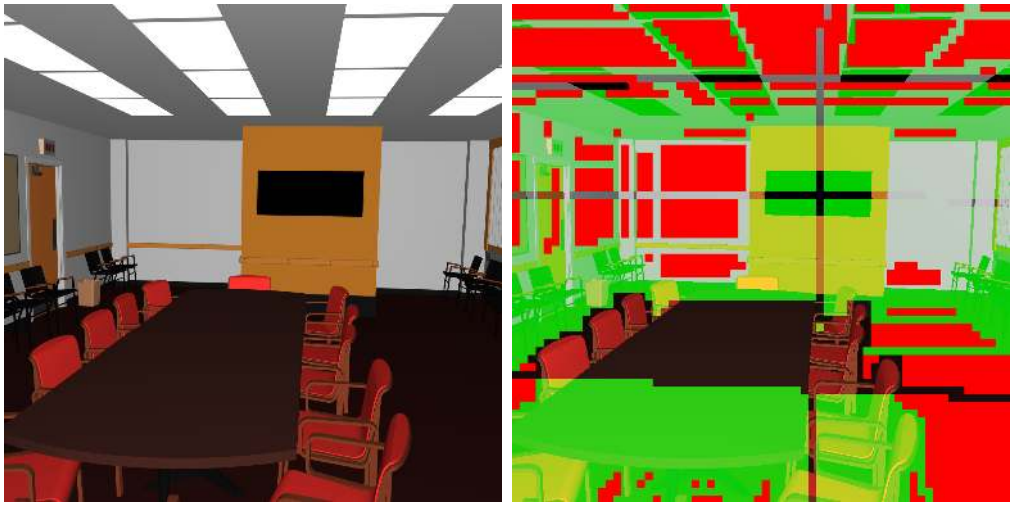


*Figure 4.18: Left Image: Original conference scene with simple shading, Right Image: Visualization of kd-tree entry points. All primary rays corresponding to the green shaded regions take advantage of kd-tree entry points which are deeper than the root of the tree. The red colored regions refer to entry points which directly correspond to a single kd-tree leaf. In this case, only intersection operations need to be applied and initial operations, e.g. traversal initialization or ray clipping can be omitted, saving many unnecessary operations.*

ray bundles exploit deep kd-tree entry points and even for a fairly large number the entry points correspond directly to kd-tree leaf nodes. In this case, no traversal but only intersection operations have to be performed. If this case is applied prior to the actual ray bundle traversal, all initial operations, e.g. ray clipping or traversal initialization can be omitted, thus saving many unnecessary operations. As this optimization technique is rather special, it is not included in any upcoming measurement.

For scenes with high occlusion the effect of using deep kd-tree entry points can even save more than half of the traversal steps. Reshetov [Reshetov05] showed that for such scenes, up to 2/3 of all traversal steps can be omitted. On the other hand, complex scenes with less occlusion, such as the VW Beetle scene, are not well-suited for the entry point algorithm. In this case, only a traversal step saving of 20% to 30% is achieved (see Table 4.2).

Table 4.2 also provides statistics for replacing the standard $4 \times 4$ bundle traversal algorithm ($XP$) by a version that is similar to the extremal traversal algorithms ($XP4$), known from the kd-tree entry search (see Figure 4.13).

The main difference between standard traversal and extremal traversal algorithms is the deactivation of invalid rays through masking operations. As the extremal algorithm relies on properties of the whole ray bundle without applying any masking operations, an overhead in traversal and intersection operations is caused. On the other hand, the advantage of extremal traversal is its simpler and faster traversal implementation as compared to the standard algorithm. Standard traversal bases the traversal order (left, right, or both children) on the decision of all active rays within a $4 \times 4$ bundle, whereas extremal traversal relies only on the decision of the ray frustum. If the introduced traversal and intersection overhead is sufficiently small, e.g. for simple scenes a performance gain of up to $30 - 40\%$ can be achieved.

| Scene | Conference | VW Beetle | Soda Hall |
|---|---|---|---|
| T / Bundle XP | 39.65 | 36.81 | 29.64 |
| I / Bundle XP | 6.03 | 5.61 | 3.68 |
| T / Bundle XP4 | 42.13 | 42.01 | 30.08 |
| I / Bundle XP4 | 7.93 | 7.06 | 5.01 |
| T / Bundle EP | 3.37 | 3.56 | 2.94 |
| T / Bundle XP (EP) | 20.46 | 25.03 | 23.27 |
| I / Bundle XP (EP) | 6.04 | 5.61 | 4.24 |
| T / Bundle XP4 (EP) | 22.99 | 30.26 | 23.89 |
| I / Bundle XP4 (EP) | 7.94 | 7.16 | 5.6 |
| Reduction T XP / XP (EP) | 48.39% | 32% | 21.49% |
| Reduction T XP4 / XP4 (EP) | 45.43% | 27.97% | 20.59% |
| Increase T XP (EP) / XP4 (EP) | 12.37% | 20.89% | 2.66% |
| Increase I XP (EP) / XP4 (EP) | 31.45% | 27.63% | 32.07% |

*Table 4.2: 1.) Average traversal steps (T / Bundle) and intersections (I / Bundle) per $4 \times 4$ primary ray bundle using standard ray bundle traversal (XP) with masking, and extremal traversal (XP4) using a ray frustum spawned by four corner rays. 2.) Additional traversal steps required for kd-tree entry point search (T / Bundle EP). Average traversal steps and intersections for standard (XP (EP)) and extremal (XP4 (EP)) per $4 \times 4$ ray bundle using kd-tree entry points. 3.) For scenes with high occlusion and less geometric complexity (Conference), up to 48% of all traversal steps are saved. For scenes with less occlusion and high complexity (VW Beetle or Soda Hall), only 32% to 21% are saved. Relying on extremal traversal (XP4) results in an increased number of intersections (up to 32%).*

For all example scenes, the overhead in traversal steps for the extremal traversal (*XP4*) as compared to the standard traversal (*XP*) varies from 2% to 20%. Morover, the extremal traversal requires roughly 30% more intersections. The overhead for intersection is larger than that for traversal because the additionally accessed kd-tree nodes, which are deeper within the kd-tree, are likely to not contain empty-space. Unfortunately, an intersection operation is more expensive than a traversal operation, especially more expensive than an extremal traversal operation. It might be beneficial to follow the idea of Reshetov [Reshetov05], who proposed (for rather incoherent rays) to apply extremal traversal in the upper 2/3 of the kd-tree and for the lower 1/3 the standard traversal with masking. This would ensure fast traversal in higher regions of the kd-tree, where the impact of extremal traversal is not that significant, and slower but more accurate traversal in the deeper kd-tree regions to prevent unnecessary operations.

| Scene | Conference |
|---|---|
| XP Bundles | 65536 |
| EP Bundles $64 \times 64$ | 224 |
| EP Bundles $16 \times 16$ | 3955 |
| T Steps EP $64 \times 64$ | 62.24 |
| T Steps EP $16 \times 16$ | 40.76 |
| T Steps EP / XP Bundle | 2.61 |

*Table 4.3: Number of primary ray bundles (frustum spawned by four corner rays) used for kd-tree point entry search (EP) and the related average traversal steps per bundle. Note that the kd-tree entry point search does not perform any intersection operations (in this implementation). Therefore, costs of 2-3 additional traversal steps per standard* $4 \times 4$ *bundle (XP) are negligible.*

Preliminary tests with a simple depth-based criteria have shown that the traversal overhead of extremal traversal can be split by half. For scenes with high occlusion, e.g. the conference scene, even pure extremal traversal can be beneficial. Especially for simple scenes with high occlusion, the faster implementation of extremal traversal outperforms the standard traversal algorithm. Note that the pure extremal traversal will not perform well with complex scenes having less occlusion, e.g. the VW Beetle scene. As no masking of inactive rays can be applied during traversal, extremal traversal will access more leaves and will therefore perform more intersection operations. The intersection overhead (typically more than 30%) is likely to offset any traversal benefit.

The overhead for extremal traversal largely depends on the underlying kd-tree. If the kd-tree is not constructed well, e.g. without applying surface area heuristics (SAH), the resulting amount of required traversal and intersection operations can be tremendous [Havran01, Wald04]. Even in the case of SAH-based kd-tree construction, it is highly beneficial to further optimize the kd-tree for extremal traversal, e.g. by avoiding many extreme small voxels. Reshetov et al. [Reshetov05] proposed several techniques for optimizing SAH-based kd-tree construction, which could additionally lower the traversal and intersection overhead.

In terms of entry search complexity, only $2-3$ additional traversal steps per $4 \times 4$ bundle have to be applied (see Table 4.3). Note that the entry point search does not perform any intersection operations, so the costs for kd-tree entry search depend exclusively on the traversal costs. The costs for applying these $2-3$ additional traversal steps per $4 \times 4$ bundle are rather negligible, because these steps are significantly faster than standard traversal steps.

As the actual ray tracing performance does not only depend on the chosen traversal algorithm but also on the chosen intersection algorithm, all total performance statistics for the triangular example scenes are postponed until Chapter 5.

### 4.4.3 Traversal Costs

Measuring the average cost including memory access for a single traversal step of a $4 \times 4$ ray bundle while using the ray-segment code from Section 4.2.3, yields 130 to 220 cycles. The first value was measured on one of the latest EM64T architectures with 16 SSE registers, while the second value was measured on a standard 32-bit processor with only 8 SSE registers. This illustrates the importance of holding the majority of relevant data within processor registers as opposed to loading the data from memory. Besides a larger number of registers, the latest processor generations offer additional features such as improved branch prediction or lower instruction latency which also helps.

The observation of improved performance on the latest architectures holds also true for the kd-tree entry search algorithm of Section 4.3. Here, the cycle costs vary between 70 and 180 cycles for a single traversal step. The less complex code results directly in lower cycle costs, and can benefit from a larger number of registers in the same way as the standard $4 \times 4$ traversal.

Amortizing the cycle costs for the standard $4 \times 4$ ray bundle traversal over the number of rays yields 8.13 to 13.75 cycles, which is very low compared to the traversal costs of 30 cycles for single rays [Wald04]. If a $4 \times 4$ ray bundle is

traversed using the extremal traversal, the amortized costs are 4.38 to 11.25.

All these cycle costs should not be seen as exact calculations but rather as rough estimates. Given the varying costs of memory access and out-of-order execution, an exact timing is very difficult to achieve.

### 4.4.4 Efficient Mailboxing

Due to the fact that a reference to the same primitive can exist in multiple kd-tree leaves, a ray bundle could perform the primitive intersection test multiple times. A simple technique to avoid such unnecessary intersection tests is *Mailboxing* [Amanatides87, Glassner89, Kirk91].

Mailboxing allows for checking if a given primitive has already been intersected by the current ray bundle or not. Therefore, mailboxing requires that a unique ID be assigned to each ray bundle. After an intersection test, the primitive is marked as already intersected by assigning the current ray bundle ID to the primitive.

Unnecessary tests can now be avoided by performing a simple check before every potential primitive intersection: If the current ray bundle ID matches the ID assigned to the primitive candidate, an intersection test between the ray bundle and the primitive has already been performed and can therefore be omitted.

| Scene | Conference | VW Beetle | Soda Hall |
|---|---|---|---|
| I Ops/Bundle without MB | 6.99 | 8.17 | 4.25 |
| I Ops/Bundle with MB | 5.97 | 5.45 | 3.68 |
| Reduction | 14.59% | 33.29% | 13.41% |

*Table 4.4: Average number of intersection tests per* $4 \times 4$ *ray bundle with and without standard mailboxing. Applying mailboxing can save* $13 - 33\%$ *of intersection tests. If the cost for intersection is significantly larger than the cost of traversal, the reduced amount of intersection tests can largely enhance ray tracing performance.*

Table 4.4 shows that for the three example scenes, mailboxing allows for reducing the amount of intersection tests per bundle between $13-33\%$, which directly corresponds to the amount of multiple intersections. The impact of mailboxing depends largely on the occlusion within the scene and the quality of the corresponding kd-tree.

As an additional ray bundle ID is required per primitive, the standard mailboxing requires a separate look-up table with a size equal to the number

```
unsigned int mbox[NUMBER_OF_PRIMITIVES];

inline void MarkIntersection(const unsigned int triId,
                             const unsigned int rayId)
{ mbox[triId] = rayId; };

inline bool AlreadyIntersected(const unsigned int triId,
                               const unsigned int rayId)
{ return mbox[triId] == rayId; };
```

*Figure 4.19: An example implementation for standard mailboxing. The 'mbox' look-up table stores for each primitive the last ray bundle ID ('rayId') with which an intersection test was performed. Prior to a potential intersection, the primitive ID is taken to determine the entry location and a comparison between the current and the stored ray bundle ID is performed. If these two IDs match, an intersection with this primitive has already been performed and can therefore be omitted.*

of primitives. Figure 4.19 shows an example implementation for standard mailboxing. Unfortunately, this approach is suboptimal for current CPU architectures because the mailbox entries are accessed at almost random order, introducing a lot of L2 cache misses. A $1,000,000$ primitive scene requires, for example, a mailbox look-up table of $1,000,000 * 4 = 3.81$ MB, which is larger than any L2 cache. Only a fraction of the complete mailbox can be held in the L2 cache and the probability that only the cached mailbox entries are accessed is rather low. Besides the actual cache misses, loading entries out of the mailbox table into the L2 cache overwrites potential kd-tree or any other required data. This can implicitly affect performance of other components of the ray tracing algorithm.

Given the problematic cache behavior a more compact mailbox layout is beneficial. Fortunately, statistical analyses have shown that a look-up table with a size equal to the number of primitives is not mandatory because a ray bundle typically accesses only a subset of all primitives during ray traversal. Therefore, a significantly reduced mailbox look-up table combined with a hashing function has proven superior. The mailbox entry is found by building a hash value with the primitive ID. Each mailbox entry stores the ray bundle and primitive ID of the last intersection together. Prior to a potential intersection, the ray bundle ID and the primitive ID are tested for equality. Figure 4.20 shows an example implementation for *hashed* mailboxing. Choosing a hash table size of a power of two allows for implementing the hash function as a simple logical *and* operation.

```
#define HASH_TABLE_SIZE 64
#define HASH_TABLE_MASK HASH_TABLE_SIZE-1

struct Entry
{
  unsigned int triId, rayId;
};

Entry mbox[HASH_TABLE_SIZE];

inline bool AlreadyIntersected(unsigned int triId, unsigned int rayId)
{
  return mbox[triId & HASH_TABLE_MASK].triId == triId &&
         mbox[triId & HASH_TABLE_MASK].rayId == rayId;
};

inline void MarkIntersection(unsigned int triId, unsigned int rayId)
{
  mbox[triId & HASH_TABLE_MASK].triId = triId;
  mbox[triId & HASH_TABLE_MASK].rayId = rayId;
};
```

*Figure 4.20: An example implementation for hashed mailboxing. Each mailbox entry stores the ray bundle and primitive ID together. The mailbox entry is determined by hashing the primitive ID. As the size of the hash table is a power of two, the table entry is determined using a simple and fast logical* and *operation. For testing if a primitive has already been intersected by the current ray bundle, both the ray bundle ID and the primitive ID have to be compared with the values of the mailbox entry.*

The efficiency of hashed mailboxing depends on the number of hash collisions during the traversal of a ray bundle. Collisions occur when the hash values of different primitive IDs map to the hash table entry. Fortunately, the primitive IDs accessed during traversal are almost randomly distributed, which results in an almost equal hash distribution. Therefore, only the size of the hashed mailbox influences the number of collisions.

Table 4.5 illustrates the average number of intersection tests per bundle using hashed mailboxing with a varying hash table size. For all test scenes, a hashed mailbox with only 64 or 128 entries provides almost the same number of intersection tests per $4 \times 4$ ray bundle ratio as a standard full-sized mailbox, while only requiring $0.5 - 1$ KB storage space. Because of an improved L2 cache behavior a hashed mailbox has been used as the default mailbox for all measurements.

| Scene | Conference | VW Beetle | Soda Hall |
|---|---|---|---|
| Standard Mailbox | 5.97 | 5.45 | 3.68 |
| Hashed Mailbox 16 Entries | 6.14 (2.8%) | 6.02 (10.4%) | 3.79 (2.9%) |
| Hashed Mailbox 32 Entries | 6.06 (1.5%) | 5.77 (5.8%) | 3.76 (2.1%) |
| Hashed Mailbox 64 Entries | 6.03 (1.0%) | 5.61 (2.9%) | 3.73 (1.3%) |
| Hashed Mailbox 128 Entries | 6.00 (0.5%) | 5.52 (1.2%) | 3.72 (1.0%) |

*Table 4.5: Average number of intersection tests per* $4 \times 4$ *ray bundle using hashed mailboxing with a varying hash table size. A hashed mailbox with only 64 or 128 entries ensure almost the same ratio (difference of less than 3%) as that of a standard full-sized mailbox. As a hashed mailbox with 128 entries requires only 1 KB storage space, the probability of a cache miss when accessing a mailbox entry is significantly lower than that of a standard mailbox.*

### 4.4.5 Shading Coherent Rays

Supporting the efficient bundling of coherent rays in a standard recursive ray tracer can be very difficult. Tracing bundles is not problematic for primary rays because those are usually generated in a coherent manner. Also, the coherence of shadow rays traced from the first hit point to a point light is usually very high. The problem appears when handling secondary rays, especially recursive reflected or refracted rays. For these kinds of rays, coherence between rays may be lost. Obviously, one could trace a bundle of incoherent rays. The problem is that with incoherent bundles only a few rays will be active during traversal, which is likely to offset the performance benefit of tracing bundles. Therefore, some kind of efficient regrouping will be necessary.

In standard ray tracing systems the recursive evaluation of secondary rays within a freely programmable surface shader is very common. A regrouping operation for secondary rays is difficult to realize efficiently in software, because a recursive shader evaluation must be temporarily stopped in order to insert a ray into the appropriate ray bundle, and it must be restarted once the bundle has been traced.

One possible solution to this problem is to split the shading model into a programmable and a fix function part. The programmable part defines a ray query and transfers the query to fix part. The queries for a complete bundle are gathered and the fix function part of the shaders traces them together as a bundle. This approach only works for secondary rays which are spawned by the same surface shader [Benthin03].

Woop et al. [Woop05] proposed to handle different shaders for a given set of intersections in a multi-pass way on a ray tracing hardware architecture. For each shader type, an individual pass is performed, excluding intersections of different shaders by masking. This approach does not include an explicit regrouping of rays because it only groups and traces rays from the same shader in a bundle (even if the rays are incoherent). The pure hardware approach makes it even possible to stop and restart recursive shader evaluation without any performance losses. Even though the same approach could also be realized in terms of software, it is unclear how effective a software implementation would be.

Even though several approaches for shading coherent rays exist, for a pure software system, an efficient algorithm for the regrouping of coherent rays and the design of an efficient shading model for ray bundles remains an open question to this day.

## 4.5 Conclusions and Future Work

In summary, current CPU architectures allow for efficient kd-tree traversal if the corresponding implementation has been carefully adapted to the underlying hardware. However, traversal implementation will be even faster if future CPU architectures offer a higher number of registers, less deep execution pipelines to lower the impact of branch mis-predictions, as well as an improved instruction set.

For the future, it will be very interesting to minimize the traversal and intersection overhead for extremal traversal. If the overhead can be minimized to a similar level as for standard traversal, extremal traversal will be the method of choice in terms of performance. Besides faster implementation, the costs for extremal traversal are constant because they are independent of the number of rays within the bundle. This will be beneficial in the case large bundles of coherent rays need to be traced, e.g. for anti-aliasing by super-sampling.

# Chapter 5

# Coherent Ray Tracing for Triangular Surfaces

Triangles are essentially the standard geometric representation for rendering in computer graphics. As triangles are the standard primitive for today's graphics hardware, scene complexity is predominantly measured in triangles. Typical scene complexity for models used in the context of virtual prototyping, ranges from several hundred thousands to millions of triangles.

As shown in Chapter 4, tracing coherent ray bundles instead of single rays is the key factor for increasing traversal performance. SSE instructions, which allow for efficient traversal implementation in the context of ray bundles, can also be applied to speed up triangle intersection.

In order to illustrate ray-triangle intersection algorithms in the context of coherent ray bundles, a small collection of triangular scenes have been used (see Chapter 4).

Researchers proposed many algorithms for computing the intersection between a single ray and a triangle [Glassner89, Badouel92, Erickson97, Möller97, Shoemake98, Wald04]. Some of these algorithms require only the vertices of the triangle as input data, others rely on extra precomputed data to ensure maximum performance. In the following sections, the implementations of two of these algorithms are presented in detail in the context of ray bundles.

This chapter starts with a presentation of the algorithm proposed by Wald et al. [Wald04] in Section 5.1. In Section 5.2, a variation of the Pluecker test is proposed which has been adapted in order to handle ray bundles efficiently. Finally, Section 5.3 illustrates the effective combination of fast kd-tree traversal and intersection for ray tracing triangular scenes.

```
struct TriAccel
{
  // plane:
  float n_u; // normal.u / normal.k
  float n_v; // normal.v / normal.k
  float n_d; // constant of plane equation
  int k;     // projection dimension

  // line equation for line ac
  float b_nu;
  float b_nv;
  float b_d;
  int pad0; // pad to 48 bytes for cache alignment purposes

  // line equation for line ab
  float c_nu;
  float c_nv;
  float c_d;
  int pad1; // pad to 48 bytes for cache alignment purposes
};
```

*Figure 5.1: Structure for storing the preprocessed triangle data. It contains triangle data that is projected onto one of the three axis-aligned planes (the corresponding dimension is indicated by parameter 'k'). The data block related to normal data is used for computing the intersection distance while the two other blocks are used for computing the barycentric coordinates. The padding to a 48-byte size allows for a better cache access pattern.*

## 5.1 Triangle Intersection I

A fast ray-triangle intersection test which can be efficiently implemented using SSE instructions was proposed by Wald et al. [Wald04]. The basic idea behind this test is to project the triangle into one of the three axis-aligned planes and to perform the computation of the barycentric coordinates in 2D instead of 3D. This allows for reducing the number of operations required. In order to ensure maximum performance, the algorithm relies on a small amount of precomputed data (see Figure 5.1) for every triangle. Relying on this precomputed data basically avoids additional instructions for data rearrangement, which can be costly using SSE.

The size of the precomputed data per triangle corresponds to 48 bytes, resulting in a tolerable amount of 45.78 MB of additional preprocessed data for a typical one-million-triangle scene. Additionally, the size of 48 bytes

corresponds to 1.5 cache-lines for a 32-byte cache-line size, or to 0.75 cache-lines for a 64-byte cache-line.

### 5.1.1 Implementation

Adapting the original algorithm [Wald04] for supporting four rays in parallel using SSE instructions is straightforward. Figure 5.2 shows an example implementation for four rays. An implementation for $4 \times 4$ rays can be simply achieved by sequentially performing the intersection code four times.

A closer look at the code shows that extra shuffle operations are required to copy a single float value (from the corresponding preprocessed *TriAccel* structure) to all SSE register elements at the beginning. Replicating the TriAccel elements four times would allow for a more suitable access but would also result in a four times larger structure size. Additionally, this would significantly increase the required memory bandwidth and lower the effect of caches, which is is not beneficial.

The first sequence of instructions computes the distances $f$ between the ray origins and the triangle plane using the precomputed normal data in the TriAccel structure. As the distance computation involves a costly division and the subsequent instructions depend on the result, the implementation uses a Newton-Raphson iteration for computing the inverse. This allows for reducing latency and for better interleaving of surrounding instructions (increased instruction level parallelism). The slightly less accurate result compared to a real division is negligible.

Comparing the four distances $f$ with the previous intersection distances forms the first potential exit point. Note that the exit is only taken if the comparison holds true for all four rays. This is an example for decision coherence. The higher the coherence of the rays, the higher the probability that all rays leave the code sequence at a given exit point.

The subsequent instructions basically compute the barycentric coordinates and compare the results to the valid bounds for a triangle. The second and third exit points require the results of the barycentric coordinate computation and are therefore located in the middle and at the end of the algorithm. After the fourth, and final exit point the data for the four intersections is updated according to the accumulated final mask.

A size of 48 bytes and a data alignment that is a multiple of the cache-line size ensures that for accessing all required data only one cache-line or only a maximum of two cache-lines (per triangle) must be loaded. A detailed discussion about the algorithmic details and the required preprocessing for the intersection algorithm is given in [Wald04].

```
TriAccel &acc = accel[triID];
static unsigned int modulo[] = {0,1,2,0,1};
const unsigned int k = acc.k;
const unsigned int ku = modulo[k+1];
const unsigned int kv = modulo[k+2];
sse_t nd =_mm_add_ps(_mm_mul_ps(_mm_set_ps1(acc.n_u),ray4.direction.t[ku]),
                    _mm_mul_ps(_mm_set_ps1(acc.n_v),ray4.direction.t[kv]));
sse_t f =_mm_add_ps(_mm_mul_ps(_mm_set_ps1(acc.n_u),ray4.origin.t[ku]),
                    _mm_mul_ps(_mm_set_ps1(acc.n_v),ray4.origin.t[kv]));
nd = _mm_add_ps(nd,ray4.direction.t[k]);
const sse_t rcp = _mm_rcp_ps(nd);
nd = _mm_sub_ps(_mm_add_ps(rcp,rcp),_mm_mul_ps(_mm_mul_ps(rcp,rcp),nd));
f  = _mm_add_ps(f,ray4.origin.t[k]);
f  = _mm_sub_ps(_mm_set_ps1(acc.n_d),f);
f  = _mm_mul_ps(f,nd);
sse_t mask = _mm_and_ps(_mm_cmpge_ps(_mm_load_ps(hit4.dist),f),
                        _mm_cmpgt_ps(f,sse_eps));
if (_mm_movemask_ps(mask)==0) continue; // -- first exit point --
const sse_t hu = _mm_add_ps(ray4.origin.t[ku],
                            _mm_mul_ps(f,ray4.direction.t[ku]));
const sse_t hv = _mm_add_ps(ray4.origin.t[kv],
                            _mm_mul_ps(f,ray4.direction.t[kv]));
// -- 'lambda' stores first barycentric coordinates --
sse_t lambda = _mm_add_ps(_mm_mul_ps(hu,_mm_set_ps1(acc.b_nu)),
                          _mm_mul_ps(hv,_mm_set_ps1(acc.b_nv)));
lambda = _mm_add_ps(lambda,_mm_set_ps1(acc.b_d));
mask = _mm_and_ps(mask,_mm_cmpgt_ps(lambda,_mm_setzero_ps()));
if (_mm_movemask_ps(mask)==0) continue; // -- second exit point --
// -- 'mue' stores second barycentric coordinates --
sse_t mue = _mm_add_ps(_mm_mul_ps(hu,_mm_set_ps1(acc.c_nu)),
                       _mm_mul_ps(hv,_mm_set_ps1(acc.c_nv)));
mue = _mm_add_ps(mue,_mm_set_ps1(acc.c_d));
mask = _mm_and_ps(mask,_mm_cmpgt_ps(mue,_mm_setzero_ps()));
if (_mm_movemask_ps(mask)==0) continue; // -- third exit point --
// -- compute 1 - 'lambda' - 'mue' --
const sse_t finalMask = _mm_and_ps(_mm_cmple_ps(_mm_add_ps(lambda,mue),
                                                sse_one),mask);
const unsigned int intFinalMask = _mm_movemask_ps(finalMask);
if (intFinalMask==0) continue; // -- fourth exit point --
// => hit4 updates ...
```

*Figure 5.2: Implementation of the triangle test by Wald et al. [Wald04] for testing four rays against one triangle using SSE instructions. The algorithm relies on a small structure of precomputed triangle data. Therefore, it requires only a small number of instructions.*

### 5.1.2 Discussion

If one has a closer look at the algorithm from Figure 5.2, it becomes evident that the intersection algorithm discards a triangle on the basis of two tests:

**Distance Test:** The test checks if the potential intersection distance is in a valid range. If for all rays, the intersection distance to the triangle plane is greater than the last valid intersection distance, the triangle can be excluded from further processing.

**Inside Test:** The test checks if the rays within a bundle pierce the triangle at all. For the algorithm shown in Figure 5.2, this is done by computing the barycentric coordinates and testing them to determine whether they are in a valid range. If the barycentric coordinates for all rays are outside the valid range, the triangle can be excluded from further processing.

The inside test takes place very late in the algorithm (essentially at the last exit point), compared to the distance test. Table 5.1 shows the average probabilities for the distance and inside test, together with the probabilities for a full test execution. All test scenes are rendered at a resolution of $1024 \times 1024$ and with a bundle size of $4 \times 4$ rays. A given ray bundle is divided into four-ray bundles, each four-ray bundle being processed individually.

Only 20 % to 28 % of all ray bundle tests require a full execution of the test. Moreover, 52 % to 68 % of all tests leave execution at the exit points related to the inside test, while only roughly 18 % (or less) exit at the distance test.

| Scene | Conference | VW Beetle | Soda Hall |
|---|---|---|---|
| Distance Test Exit % | 17.99 | 13.05 | 19.30 |
| Inside Test Exit % | 61.98 | 68.90 | 52.32 |
| Full Test % | 20.03 | 18.05 | 28.38 |

*Table 5.1: Average probability for exiting at the distance and inside test as well as for performing the full test execution. The three example scenes are rendered at a resolution of* $1024 \times 1024$ *with 16 rays per bundle, casting only primary rays. Approximately 20 % to 28 % of all four-ray bundle intersection tests require a full test execution. Between 52 % and 68 % of all tests exit at the inside test and therefore miss the triangle completely.*

Moreover, a certain amount of the early exits performed by the distance test can be handled by the inside test, as shown in Table 5.2. Therefore, the

actual number of full test executions is only moderately increased. Given these ratios, it would be beneficial to perform (if possible) the inside test in a first step, and the distance test as a second step.

| Scene | Conference | VW Beetle | Soda Hall |
|---|---|---|---|
| Full Test (with DT) % | 20.03 | 18.05 | 28.38 |
| Full Test (without DT) % | 26.26 | 20.64 | 32.76 |

*Table 5.2: Probability of full test execution with and without a distance test. Removing the distance test results in an only moderately increased number of full test executions.*

## 5.2 Triangle Intersection II

An intersection test which is able to quickly perform the inside test first is the so-called *Pluecker test* [Erickson97, Shoemake98]. Instead of using barycentric coordinates for the inside test, the Pluecker test relies on testing the relations between a ray and the triangle edge(s).

The Pluecker test takes advantage of the properties of *Pluecker coordinates* [Shoemake98], which will be briefly described in this section. Each directed line which uses 3D coordinates can be represented in 6D Pluecker space. Given two 3D points $A$ and $B$, the corresponding line $L$ in Pluecker space is defined as $L = [A-B, A \times B]$, where $\times$ refers to a cross product. For a direction $D$ and an origin $O$, the line $L$ is defined as $L = [D, D \times O]$. A very useful operation concerning the relation of lines in Pluecker space is the so-called *inner-product*. The inner-product defines how two lines pass each other in space. For two lines $L_0 = [U_0, U_1]$ and $L_1 = [V_0, V_1]$, the inner-product is defined as $L_0 * L_1 = U_0 * V_1 + U_1 * V_0$, where $*$ refers to a standard dot product. If the result equals 0 the lines intersect, while a negative (positive) result indicates that the lines pass clockwise (counterclockwise).

A triangle intersection test using Pluecker coordinates is straightforward. By expressing the ray and all three triangle edges as Pluecker lines, a ray-triangle intersection criteria is defined as follows: a ray intersects the triangle if the inner-products between the ray and all edges have the same sign. Note that the inner-product uses only multiplications and additions, allowing for an efficient implementation by SSE instructions.

In the following, a slightly modified version of a Pluecker test is presented. Assuming all rays within the ray bundle share a common origin, one can

further optimize the original algorithm [Shoemake98]. Through transferring the ray origin to the origin of the coordinate system (by subtracting the origin $O$ from the triangle vertices $A$ and $B$), the inner-product between the ray $R = [D, D \times O]$ and the triangle edge $E = [A - B, A \times B]$ is simplified to

$$\begin{aligned} R * E &= D * ((A - O) \times (B - 0)) + D \times (O - O) * ((A - O) - (B - O)) \\ &= D * ((A - O) \times (B - 0)) + 0 * (A - B) \\ &= D * ((A - O) \times (B - 0)) \end{aligned}$$

Having a common origin per ray bundle, e.g. for primary rays or for shadow rays to a point light, allows for efficiently amortizing the computation costs for $(A-O) \times (B-0)$ (and the required computations for the remaining two edges) over all rays within the bundle. The actual inside test for a four-ray bundle can be performed simply by evaluating three dot products using SSE instructions.

In order to obtain the intersection distances $t$, the following equation has to be evaluated after the edge tests:

$$t = \frac{N * (A - O)}{D * N}$$

where $N$ is the triangle normal. As $N * (A-O)$ is fixed for the whole bundle it can also be precomputed. Note that the normal $N$ of each triangle can be precomputed and stored as preprocessed data during scene definition. On the other hand, one can omit the preprocessing and compute all necessary data on the fly. In terms of changing vertices, e.g. for dynamic animation, this approach can be more efficient than carrying out an extra preprocessing step.

### 5.2.1 Implementation

A simplified implementation of the intersection algorithm is shown in Figure 5.3. The *Init* function is used for precomputing all edge and normal data for the whole bundle. The actual intersection code starts with the computation of three dot products. The corresponding signs for all three edges are compared for equality. If no ray indicates a valid intersection, no bit is set in the *mask* value, and the next ray bundle is processed. If at least one ray has a valid intersection, the execution continues with the computation of intersection distances. The look-up into the *sseMaskTable* table loads a

```
// -- precomputed per bundle --
SSEVec4 v0_cross, v1_cross, v2_cross, normal;
sse_t nominator, triNum;

inline void Init(const R3 &a,const R3 &b,const R3 &c,
                 const R3 &origin,int id)
{
 v1_cross = (b-origin)^(a-origin);
 v2_cross = (a-origin)^(c-origin);
 v0_cross = (c-origin)^(b-origin);
 const R3 n = (c-a)^(b-a); normal = n; // convert R3 -> SSEVec4
 nominator = _mm_set_ps1(n * (a-origin));
 triNum = _mm_set_ps1(*(float *)&id);
}
inline void Intersect(const SSERay4 *ray4,SSEIntersection4 *hit4)
{
 for (unsigned int i=0;i<BUNDLES;i++) {
   const sse_t v0d = Dot(v0_cross,ray4[i].direction);
   const unsigned int v0s = _mm_movemask_ps(v0d);
   const sse_t v1d = Dot(v1_cross,ray4[i].direction);
   const unsigned int v1s = _mm_movemask_ps(v1d);
   const sse_t v2d = Dot(v2_cross,ray4[i].direction);
   const unsigned int v2s = _mm_movemask_ps(v2d);
   const unsigned int mask = (v0s & v1s & v2s) |
                             ((v0s^0xf) & (v1s^0xf) & (v2s^0xf));
   if ( mask > 0 ) {
    const sse_t dist = _mm_mul_ps(nominator,
                                  Inverse(Dot(ray4[i].direction,normal)));
    sse_t finalMask = *(sse_t*)&sseMaskTable[mask][0];
    const sse_t vol = Inverse(_mm_add_ps(_mm_add_ps(v0d,v1d),v2d));
    finalMask = _mm_and_ps(_mm_cmple_ps(dist,*(sse_t*)hit4[i].dist),
                                        finalMask);
    finalMask = _mm_and_ps(_mm_cmpge_ps(dist,EPSILON),finalMask);
    const sse_t lambda = _mm_mul_ps(v0d,vol);
    const sse_t mue = _mm_mul_ps(v2d,vol);
    // => update intersection structure: hit4[i]
   }
 }
}
```

*Figure 5.3: A reliable triangle intersection test based on Pluecker coordinates adapted for ray bundles with a common origin. A triangle can be pruned early by performing only three dot products (one for each edge) using precomputed data. The cost for preprocessing Pluecker coordinates for all three triangle edges is amortized over the ray bundle.*

binary SSE mask. Each of the four lowest bits is replicated 32 times in the corresponding register element: If, for example, the lowest bit of four bits is set, the lowest element of the SSE register will be set to $0xffffffff$, and to $0x00000000$ otherwise.

The results of the inner-products can also be used to obtain the barycentric coordinates. The three results correspond to the homogeneous barycentric coordinates. In order to determine the final coordinates, a division by the sum of all three coordinate values has to be performed. Note that the division does not need to be performed every time, but only once at the end of ray traversal. Note also that this requires all three coordinate values to be temporarily stored and updated.

The actual computation of intersection distances could also be done without an inverse operation. The three barycentric coordinates $u, v, w$ and the triangle vertices $VertexA$, $VertexB$, and $VertexC$ allow for easily determining the intersection point $P(u, v, w)$:

$$P(u, v, w) = VertexA * u + VertexB * v + VertexC * w$$

The intersection distance is the projection along the direction of the ray:

$$distance = ray.direction * (P(u, v, w) - ray.origin)$$

Unfortunately, the approach of computing the distance by projection along the direction is a few percent slower than computing the distance by division because of the increased number of instructions required. The major advantage of the division-free approach is that it exclusively relies on multiply-add sequences which might be beneficial for future architectures.

## 5.3 Results

Accurately determining run-time costs in CPU cycles for the different triangle intersection algorithms is difficult because of the varying early exit point probabilities. For the example scenes shown in Figure 4.17, three cost categories have been defined. The first one measures the cycles until an early exit with respect to a triangle test occurs, the second one corresponds to a full test execution, while the third one represents the average cost based on the probability distribution of the exit points for the particular view.

Table 5.3 presents the cost for all three categories with respect to a bundle of $4 \times 4$ rays. The numbers in parentheses correspond to the amortized

| Intersection Test | Miss Triangle | Full Execution | Average |
|---|---|---|---|
| Projection (cycles) | 420 (26.25) | 620 (38.75) | 480 (30) |
| Pluecker (cycles) | 310 (19.38) | 590 (36.88) | 385 (24.06) |

*Table 5.3: Triangle intersection cost in CPU cycles for different execution scenarios. All costs are measured for a bundle of* 16 *ray, the numbers in brackets correspond to the amortized costs per ray. The first number gives the cost if all rays miss the triangle, the second if all rays hit the triangle. The third takes the probability distribution of exit points into account.*

cost per ray. Note that for the Pluecker test, the cycle cost includes the preprocessing step.

If the probability that rays miss the current triangle is high, the Pluecker test is able to play off its strength. Depending on the view and scene, an average speedup of up to 25% (for the triangle test) can be achieved compared to the test by Wald et al. (see Section 5.1). Note that the total speedup of the core ray tracing performance lies typically between $2 - 10\%$, because intersection computations take only a certain part of the time (see below). However, for most scenes, the Pluecker intersection test is the fastest for coherent bundles and is therefore used as the default test for ray tracing triangular scenes.

One could further optimize the Pluecker test by not executing the test per triangle but by executing it first for all edges and later reusing the results for the triangles. If the ratio of shared triangle edges is high, a significant amount of operations can be saved.

Applying an edge-based intersection has the additional advantage of being reliable in the intersection decision. Due to numerical instabilities an intersection test which relies on the comparison of barycentric coordinates can produce wrong results at edges of adjacent triangles. In this case, the test returns a wrong miss which can result in visible artifacts. An edge-based test that uses a fixed order of edge vertices is non-ambiguous and therefore reliable.

Combining the techniques for traversal of coherent ray bundles from Chapter 4 with the fast triangle intersection tests for ray bundles allows for easily achieving interactive ray tracing performance even on a single desktop PC. Table 5.4 illustrates performance statistics using a commodity 2.2 GHz Intel Pentium-IV PC.

Tracing ray bundles allows for achieving interactive performance even on a single CPU system. Applying performance analysis with VTune to the

core ray tracing algorithm shows that roughly 60% of the time is spent on traversal, while 40% is spent on intersection tests. Good kd-tree entry points reduces the required traversal steps per bundle by up to 50%. Reducing the number of required traversal steps per bundle shifts the traversal/intersection ratio to 40% for traversal and 60% for intersection. This illustrates that the intersection test is now becoming the limiting factor. Further speed optimization should therefore concentrate on speeding up triangle intersection.

| Scene ($1024 \times 1024$) | Conference | VW Beetle | Soda Hall |
|---|---|---|---|
| Without Entry Point Search (fps) | 3.100 | 3.150 | 3.432 |
| With Entry Point Search (fps) | 4.052 | 3.556 | 3.854 |
| Speedup | 30.71% | 12.8% | 12.29% |

*Table 5.4: Performance statistics for triangular scenes rendered at a resolution of* $1024 \times 1024$ *with a ray bundle size of* $4 \times 4$*. All tests were run on a commodity 2.2 GHz Intel Pentium-IV PC, casting only primary rays. The first row illustrates performance statistics without applying kd-tree entry point search, while the second row takes advantage of entry point search. Even though the entry point search efficiently reduces the required number of traversal steps per* $4 \times 4$ *bundle by up to 50%, the total performance gain is rather small (between 12-30%). The reason for this is that the triangle intersection is more and more becoming the bottleneck: For a typical scene, 60% of the ray tracing time is taken up by traversal while 40% is taken up by intersection.*

Due to the fact that a resolution of $1024 \times 1024$ yields 65536 ray bundles of the size $4 \times 4$ and each bundle requires an average of 39.49 (conference scene) traversal steps without applying entry point search, the required memory bandwidth for kd-tree nodes is $65536 * 39.49 * 8 = 19.745$ MB/frame. Applying the kd-tree entry point algorithm cuts down the number of traversal steps to 50%, reducing the bandwidth accordingly.

For the intersection tests, the 65536 ray bundles require an average bandwidth of 6.06 times the size of the triangle intersection data. Applying the test proposed by Wald et al. [Wald04], requires 48 bytes of precomputed data per triangle. This results in a total bandwidth of $65536 * 6.06 * 48 = 18.18$ MB/frame. Therefore, the total required bandwidth of external data sums up to 37.925 MB/frame. Most of these bandwidths ($> 90\%$ percent) can be compensated for due to the multi-level cache hierarchies of current CPUs, which yields an external memory bandwidth of less than 4 MB/frame.

## 5.4 Conclusions and Future Work

Section 5.3 showed that the combination of efficient traversal, kd-tree entry point search, and triangle intersection for ray bundles allows for achieving interactive ray tracing even on a single CPU. All tests were run on a 2.2 GHz Intel Pentium-IV processor, more recent CPU versions will therefore achieve even better results.

As fast ray bundle traversal increases the impact of triangle intersection on the core ray tracing performance, a triangle intersection test for ray bundles has been proposed that efficiently prunes non-intersecting triangles. For future implementations, it might be necessary to further speed up the intersection test for coherent ray bundles.

A promising approach to this problem could be to store the triangle geometry as small index face sets instead of individual triangles. In the case a ray bundle enters a leaf, a fast side test, e.g. a Pluecker test, would first be performed for all edges. In a second step, the results of the edge tests would be passed to the actual triangle test. Given the high probability of shared edges within a kd-tree leaf, the total amount of operations per leaf could be reduced. Moreover, the side test would ensure high instruction level parallelism, because no dependencies exist between the computation of different edges. In terms of implementation, a kd-tree leaf would not only store references to triangles but additionally references to triangle edges.

# Chapter 6

# Coherent Ray Tracing for Freeform Surfaces

Almost all CAD systems and design tools rely on freeform surfaces such as B-Splines, NURBS, or subdivision surfaces for representing 3D surfaces. However, almost all commercial ray tracing systems tessellate freeform surfaces into simpler geometric primitives, e.g. triangles. The tessellation process is usually applied before the actual rendering process starts. Tessellation itself, and therefore the increased amount of scene data, is often considered acceptable because ray tracing is known for its logarithmic behavior in scene complexity.

In Chapter 5, it has been shown that interactive ray tracing on commodity hardware is possible even when dealing with millions of triangles. It is thus legitimate to ask why freeform surfaces should be ray traced directly. The answer to this question is simple: Tessellating freeform surfaces into triangles has serious drawbacks:

**Scene and Storage Complexity:** The amount of required storage space for tessellated triangles is orders of magnitudes higher than for the original freeform data.

**Accuracy:** Tessellation can be seen as producing a snapshot of the freeform data with a given accuracy. Unfortunately, the accuracy cannot easily be changed afterwards.

**Preprocessing Cost:** Tessellation itself and the preprocessing steps required later on, such as the construction of spatial acceleration structures, require a significant amount of time, slowing down the rendering workflow and thus the interaction with the scene.

As a consequence, direct ray tracing of freeform surfaces is becoming more and more important. Unfortunately, almost none of the existing approaches has ever targeted interactivity, resulting in a limitation to off-line systems.

The industrial standard representation for freeform surfaces are *NURBS* (*N*on *U*niform *R*ational *B*-*S*plines). Even though NURBS surfaces allow a very compact representation of freeform surfaces, see [Foley97], the core operations such as surface evaluation, surface subdivision, convex hull computation, etc. are very complex and costly when directly using NURBS surfaces. As these operations are essential for a ray-surface intersection test, their cost easily dominate the total intersection cost.

In order to provide simpler and less costly core operations, the original NURBS representation has been converted into a bicubic Bézier representation by performing knot insertion [Piegl97, Farin96, Foley97] and degree reduction [Piegl97, Farin96]. In the case a NURBS surface is non-rational, it can be represented by a set of Bézier surfaces of the same degree without loss of accuracy. Applying degree reduction by recursive subdivision of higher degree Bézier surfaces causes some loss of accuracy, but this is usually negligible in the context of visualization. However, as compared to triangles, Bézier surfaces are the method of choice when it comes to representing smooth surfaces.

In the context of Bézier surfaces, the term *surface* typically refers to a compound set of Bézier patches. As Bézier patches are the basic primitive, only this type will be considered in the following.

In order to achieve a compromise between flexibility, loss of accuracy, and implementation efficiency, a (fixed) patch degree of three (bicubic) for Bézier patches is chosen (see Section 6.1). As a single NURBS surface typically converts to an entire set of Bézier patches, the conversion increases the amount of data required for representing the surface. However, compared to the conversion into triangles, storing bicubic Bézier patches requires less storage space: The higher the curved shape of a patch, the more triangles are required for an accurate approximation.

The format chosen for storing the sixteen control points of a bicubic Bézier patch is the SSE-suitable SOA format (see Figure 6.1). Storing the control point matrix in the SOA format using single precision floating point values requires 192 bytes in total, which allows for effectively storing one thousand patches in less than 200 KB.

Before addressing the problem of interactive ray tracing scenes consisting of many bicubic Bézier patches, the fundamentals of Bézier curves and patches will be presented in Section 6.1. Section 6.2 to Section 6.6 will be dealing with the different approaches for a ray-patch intersection algorithm and the extension for supporting bundles of rays. Section 6.7 presents a

```
struct SSEVec4 {
  sse_t t[3]; // SOA format: xxxx, yyyy, zzzz
};

struct BicubicBezierPatch3D {
  SSEVec4 p[4];
};
```

*Figure 6.1: Storing the* 16 *control points in the SSE-suitable SOA format, requiring a total of* 192 *bytes (using single precision floating point accuracy).*

detailed comparison of all approaches. Building spatial index structures for efficiently handling scenes consisting of many Bézier patches will be the focus of Section 6.8. Section 6.9 briefly describes how trimming curves are integrated into the intersection algorithms. Finally, Section 6.10 will give performance statistics for all presented algorithms using a set of test scenes.

## 6.1 Bézier Fundamentals

The fundamental basis for Bézier curves and surfaces are the *Bernstein polynomials* [Piegl97, Farin96, Foley97]. Bernstein polynomials are defined as:

$$B_i^n(u) = \left( \begin{array}{c} n \\ i \end{array} \right) (\frac{u-a}{b-a})^i (\frac{b-u}{b-a})^{n-i} \, , u \in [a;b] \tag{6.1}$$

Usually the Bernstein polynomials are defined over the unit interval $[0,1]$ which simplifies the definition to:

$$B_i^n(u) = \left( \begin{array}{c} n \\ i \end{array} \right) u^i (1-u)^{n-i} \, , u \in [0,1] \tag{6.2}$$

These polynomials work as blending functions between the curve or patch control points. Bernstein polynomials have important properties that will translate directly to the properties of Bézier curves and Bézier patches [Piegl97, Farin96, Foley97] later on. In the following, a brief overview of the most useful properties is given:

- *Non-negative over domain:* $B_i^n(u) \geq 0 \, , u \in [a,b]$
- *Partition of unity:* $\sum_{i=0}^{n} B_i^n(u) = 1 \, , u \in [a,b]$

- *Recursive definition:* $B_i^n(u) = (\frac{b-u}{b-a})B_i^{n-1}(u) + (\frac{u-a}{b-a})B_{i-1}^{n-1}(u)$
- *Derivatives:* $(B_i^n(u))' = n(B_{i-1}^{n-1}(u) - B_i^{n-1}(u))$

For example, the Bernstein polynomials of degree three are defined by the following four equations:

$$\begin{aligned} B_0^3(u) &= \begin{pmatrix} 3 \\ 0 \end{pmatrix} u^0(1-u)^3 = (1-u)^3 \\ B_1^3(u) &= \begin{pmatrix} 3 \\ 1 \end{pmatrix} u^1(1-u)^2 = 3u(1-u)^2 \\ B_2^3(u) &= \begin{pmatrix} 3 \\ 2 \end{pmatrix} u^2(1-u)^1 = 3u^2(1-u) \\ B_3^3(u) &= \begin{pmatrix} 3 \\ 3 \end{pmatrix} u^3(1-u)^0 = u^3 \end{aligned} \tag{6.3}$$

For efficiency reasons it is not advisable to evaluate these functions via their recursive definition, but to use the compact form of Equation 6.3.

### 6.1.1 Bézier Curves

Bézier curves of degree $n$ are defined over a set of control points $\{P_i, 0 \leq i \leq n\}$ which define the shape of the curve. The basis functions of Bézier curves are the Bernstein polynomials, resulting in the following curve definition:

$$P^n(u) = \sum_{i=0}^{n} B_i^n(u)P_i \, , P_i \in \mathbb{R}^n$$

The control points $P_i$ are usually elements of $\mathbb{R}^2$ or $\mathbb{R}^3$. A single bicubic Bézier curve (see Figure 6.2) can therefore be efficiently represented by $4 * 2 * 4 = 32$ bytes using single precision floating point values. The polygon defined by the control points is called *control polygon*.

As using Bernstein polynomials are used as the basis functions, Bézier curves inherit some useful properties:

- *Convex Hull:* The inherited properties of non-negativity (and the partition of unity) cause any point of the curve to be inside the convex hull of the curve control points.
- *Interpolation of start and end point:* The curve will pass through the start and end points defining the curve, e.g. $P^n(0) = P_0$ and $P^n(1) = P_n$.

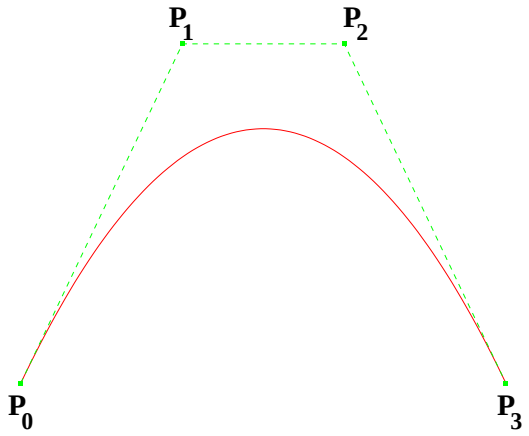



*Figure 6.2: A bicubic Bézier curve defined by four control points.*

- *Variation Diminishing Property:* A line or plane does not intersect the curve more often than the polygon defined by the control points.
- *Affine Transformation:* The curve is invariant under affine transformation (realized by transforming the control points), meaning that it does not change its shape.

The convex hull property will prove very useful later on because various intersection algorithms use it as an efficient pruning test: If a ray does not intersect the convex hull of the curve, it cannot intersect the curve itself.

Apart from their inherited properties, Bézier curves own efficient algorithms for subdivision, refinement, and evaluation. The most popular algorithm which handles all three tasks was presented by *de Casteljau* [Piegl97, Foley97]. This iterative algorithm is based on the affine combination of control points. Let

$$P_i^0 = P_i \,, 0 \leq i \leq n$$

the next set of control points is determined by

$$P_i^j = (1-u)P_i^{j-1} + uP_{i+1}^{j-1} \,, 1 \leq j \leq n, 0 \leq i+j \leq n$$

for a parameter $u \in [0,1]$. Figure 6.3 illustrates the properties of the de Casteljau algorithm: Point $P_0^n$ corresponds to the evaluation of the Bézier curve at parameter value $u$. Control points $P_0^j \,, 0 \leq j \leq n$ define the control points of the left sub-curve, while $P_i^j \,, 0 \leq j \leq n, i+j = n$ define those of the right sub-curve of the original curve subdivided at parameter $u$. Through continuous subdivision, the corresponding control polygon converges to the

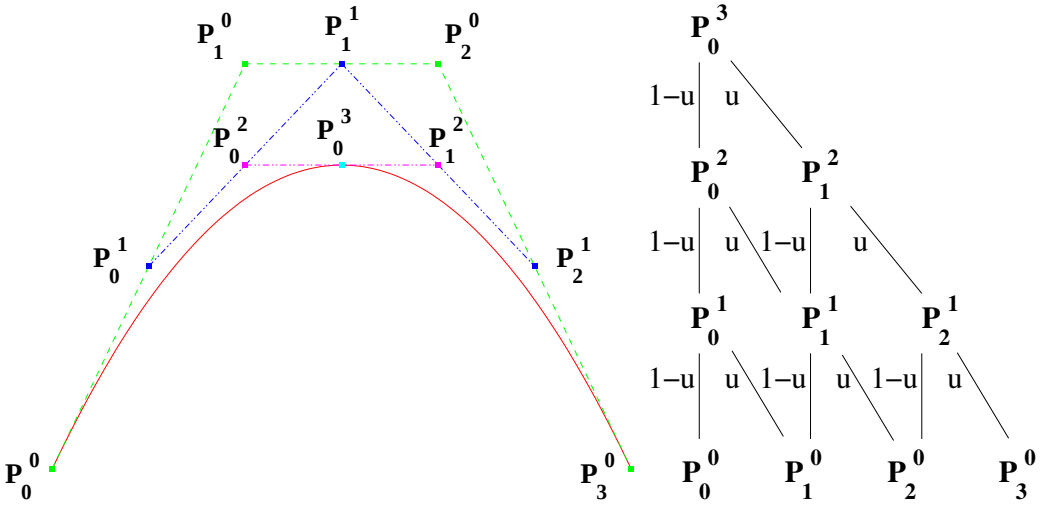



*Figure 6.3: Left: Applying the de Casteljau algorithm for a bicubic Bézier curve $P(u)$ with $u = 0.5$. $P_0^n$ corresponds to point $P(0.5)$. $P_0^j, 0 \leq j \leq 3$ defines the control points of the left sub-curve, $P_i^j, 0 \leq j \leq 3, i+j = 3$ those of the right sub-curve. Right: The de Casteljau algorithm illustrated as a triangular scheme of linear combinations.*

curve itself and can therefore be used (following a sufficient number of subdivisions) as an approximation. This process is often referred to as *refinement*.

### 6.1.2 Bézier Patches

A Bézier patch is a surface extension (tensor product) of a Bézier curve. It is constructed from $n \times m$ control points $\{P_{ij}, 0 \leq i \leq n, 0 \leq j \leq m\}$ using products of two univariate Bernstein polynomials as blending functions. A Bézier patch is defined as a two-dimensional parametric function:

$$P(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} B_i^n(u) B_j^m(v) P_{ij} \ , P_{ij} \in \mathbb{R}^n$$

A bicubic Bézier patch (see Figure 6.4) with its $4 \times 4 = 16$ control points defined in $\mathbb{R}^3$ can be efficiently represented by $16 * 3 * 4 = 192$ bytes using single precision accuracy.

Similar properties as for Bézier curves also hold for Bézier patches: $P_{00}$, $P_{0n}$, $P_{m0}$, $P_{nm}$ are on the patch surface, the patch lies in the convex hull of its control points, and the patch is invariant under affine transformation. The de Casteljau algorithm for Bézier patches works similar to the one for Bézier curves, with the exception that the algorithm can be applied to either one of the two parametric directions (see Figure 6.3).

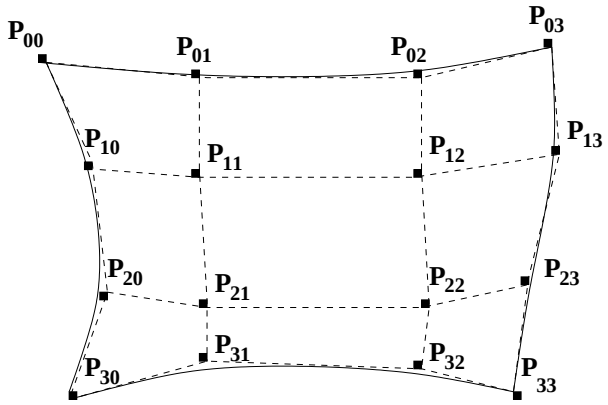



*Figure 6.4: A bicubic Bézier patch defined by its* 16 *control points. Note that* $P_{00}$,$P_{03}$,$P_{30}$, *and* $P_{33}$ *lie on the surface itself.*

In certain cases, it is useful to represent the Bézier patch in its matrix representation. For a bicubic Bézier patch, this representation is defined as:

$$P(u,v) \;=\; U^T B^T P B V \tag{6.4}$$

$$= \begin{bmatrix} u^3 \\ u^2 \\ u \\ 1 \end{bmatrix}^T B^T \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix} B \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix}$$

where the Bernstein matrix $B$ is defined as

$$B \;=\; \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \tag{6.5}$$

If the control points are known not to change, the Bernstein matrix $B$ can be multiplied to the control point matrix, resulting in

$$= \begin{bmatrix} u^3 \\ u^2 \\ u \\ 1 \end{bmatrix}^T \begin{bmatrix} P^*_{00} & P^*_{01} & P^*_{02} & P^*_{03} \\ P^*_{10} & P^*_{11} & P^*_{12} & P^*_{13} \\ P^*_{20} & P^*_{21} & P^*_{22} & P^*_{23} \\ P^*_{30} & P^*_{31} & P^*_{32} & P^*_{33} \end{bmatrix} \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix}$$

The evaluation of $P(u,v)$ through the application of the new matrix $P^*_{ij}$ requires only multiplication (from left and right) with the two vectors $U^T$ and $V$. This allows for a more efficient implementation. In the same manner, the derivatives can be determined by simply replacing the vector $U^T$ by $U'^T = \begin{bmatrix} 3u^2 & 2u & 1 & 0 \end{bmatrix}$, respectively $V$ by $V'$.

## 6.2 The Ray-Patch Intersection Problem

Testing for a valid intersection between a ray and a bicubic Bézier patch is a significantly more complex operation than a ray-triangle intersection test. In the last two decades researchers have proposed many approaches for solving the intersection problem. In the following, some of these will be briefly discussed.

Nishita et al. [Nishita90] described an iterative algorithm called *Bézier clipping* (see Section 6.6) to compute ray-patch intersections by identifying and cutting away regions of the patch, which are known not to intersect the ray. Campagna et al. [Campagna97] later solved inherent numerical issues of the original algorithm and proposed algorithmic optimizations.

Martin et al. [Martin00] presented a framework for integrating ray tracing of trimmed NURBS into existing rendering architectures. The framework can be easily transferred for handling bicubic Bézier patches instead of the original NURBS. A numerical approach called *Newton iteration* was chosen as basis for intersection computation. Newton iteration (see Section 6.4) allows for quickly converging to a solution if the starting values are sufficiently close to the correct solution. Martin et al. used a spatial subdivision structure for each patch to ensure a sufficiently close starting values.

Wang et al. [Wang01] combined Newton iteration with Bézier clipping and used the coherence of neighboring rays to speed up Nishita's original algorithm by roughly a factor of three. Unfortunately, the algorithm requires additional data structures to prevent wrong results through reusing the data of neighboring rays.

Sweeney et al. [Sweeney86] concentrated on speeding up ray tracing of B-Spline surfaces by basing the initial guess of the Newton iteration on the intersection between a ray and the sufficiently refined control mesh.

Researchers have also proposed dedicated hardware designs for ray tracing freeform surfaces. Lewis et al. [Lewis02] presented a design of a pipelined architecture for ray-patch intersection. The design is too closely related to custom hardware to be beneficial for a software approach. Furthermore, its main target was to serve as a proof of concept, and not to provide a high performance solution.

Parker et al. [Parker99b] implemented a realtime ray tracing system using a massively parallel approach (a large shared-memory supercomputer). The system also supported Bézier patches, while using Newton iteration for ray-patch intersection.

To sum it up, all approaches so far more or less have been focused on a general implementation and have never explicitly targeted high performance. Furthermore, the corresponding algorithms were neither designed nor implemented with respect to the underlying processor architecture, causing major performance penalties.

Approaches designed exclusively to the underlying hardware have been proposed by Benthin et al. [Benthin04] and Geimer et al. [Geimer05]. Both methods have been optimized to achieve interactive performance on a single CPU (see Section 6.3 and Section 6.4).

In the following, intersection algorithms considered suitable for fast implementation will be discussed in detail. Bottlenecks, performance issues and, in particular, coding guidelines for an implementation using SSE instructions will be presented. Moreover, it will be shown that, depending on the algorithm, the approach of performing intersection computation for a bundle of rays in parallel (see Chapter 5) is also beneficial in the context of bicubic Bézier patches.

## 6.3 Uniform Refinement

As a basis for the first ray-patch intersection algorithm, a simple refinement-based approach is chosen. Refinement allows for easily controlling approximation accuracy and, in particular, isolating the spatial location of the intersection. The refinement-based intersection thus relies on the following core operations:

**Pruning:** A conservative and simple pruning test is used to decide whether the ray can intersect a given patch or not.

**Refinement:** A refinement step subdivides a given patch into a set of (usually two to four) child patches.

**Final Intersection:** When reaching the maximum refinement level, an intersection test with an approximate representation of the current patch is performed.

Instead of adaptively refining the patch, a maximum predefined refinement level is used. It may seem that introducing the additional work of

```
BicubicBezierPatch3D patch; // current active patch
Stack stack;                // patch stack
int depth = 0;
while(1)
{
 while(1)
  {
   if (Prune(patch)) break;
   if (depth == MAX_DEPTH)
    {
     FinalIntersection(patch);
     break;
    }
   if (depth % 2)
     patch.RefineHorizontal(child0,child1);
   else
     patch.RefineVertical(child0,child1);
   depth++;
   patch = child0;
   stack.push(child1,depth);
  }
  if (stack.empty()) break;
  stack.pop(patch,depth);
}
```

*Figure 6.5: Pseudo C++ code for intersection based on three core operations for uniform refinement. The parametric direction for the refinement operation is determined by the current refinement depth.*

refining the patch to a fixed predefined level is not as clever as avoiding the work by adaptive refining, but it makes the previously required crack-handling and adaptivity tests obsolete. The main difficulty of this approach is to chose the right refinement level, see [Benthin04].

Figure 6.5 shows an example implementation for the complete ray-patch intersection step. Note that uniform refinement does not require additional data structures. This makes the algorithm very memory-efficient. In the following, the core operations will be discussed in detail.

### 6.3.1 Pruning Implementation

The pruning operation can be seen as the most important of all core operations, because it is the first one in order and therefore the one applied most often (a pruned patch is never refined nor intersected). To efficiently implement the pruning operation we represent the ray as the intersection of

```
inline bool Prune(const BicubicBezierPatch3D &patch,
                  const RayPlanesSSE &plane)
{
 unsigned int signsU = 0;
 for (unsigned int i=0;i<4;i++)
  {
   const sse_t d = _mm_add_ps(_mm_add_ps(_mm_mul_ps(plane.Nu.t[0],
                                                    patch.p[i].t[0]),
                                          _mm_mul_ps(plane.Nu.t[1],
                                                    patch.p[i].t[1])),
                              _mm_mul_ps(plane.Nu.t[2],patch.p[i].t[2]));
   signsU |= _mm_movemask_ps(_mm_sub_ps(d,plane.du)) << (4*i);
  }
 if (signsU == 0xffff || signsU == 0) return true;

 unsigned int signsV = 0;
 for (unsigned int i=0;i<4;i++)
  {
   const sse_t d = _mm_add_ps(_mm_add_ps(_mm_mul_ps(plane.Nv.t[0],
                                                    patch.p[i].t[0]),
                                          _mm_mul_ps(plane.Nv.t[1],
                                                    patch.p[i].t[1])),
                              _mm_mul_ps(plane.Nv.t[2],patch.p[i].t[2]));
   signsV |= _mm_movemask_ps(_mm_sub_ps(d,plane.dv)) << (4*i);
  }
 if (signsV == 0xffff || signsV == 0) return true;
 return false;
}
```

*Figure 6.6: Fast pruning test by testing the signs of control point distances to the two ray planes. If the sixteen signs match, all control points lie entirely in one half-space determined by one of the ray planes, and the corresponding patch cannot be intersected by the ray.*

two orthogonal planes [Sweeney86, Woodward89]. The distance of all control points to both planes are computed yielding 2D coordinates. If all 2D coordinates lie entirely in one half-space (with respect to a certain plane), the ray cannot intersect the corresponding patch because of the convex hull criteria.

The projection of control points can be realized via a sequence of dot products. SSE instructions allow for performing four dot product operations in parallel. For the entire patch, this results in only 12 parallel multiplications and 12 parallel additions plus 1 instruction for obtaining the sign bits for each plane (see Figure 6.6).

```
inline void RefineVertical(BicubicBezierPatch3D &patch0,
                           BicubicBezierPatch3D &patch1)
{
 static const sse_t half = _mm_set_ps1(0.5f);
 patch0.p[0] = p[0];
 patch1.p[3] = p[3];
 patch0.p[1].t[0] = _mm_mul_ps(half,_mm_add_ps(p[0].t[0],p[1].t[0]));
 patch0.p[1].t[1] = _mm_mul_ps(half,_mm_add_ps(p[0].t[1],p[1].t[1]));
 patch0.p[1].t[2] = _mm_mul_ps(half,_mm_add_ps(p[0].t[2],p[1].t[2]));
 patch1.p[2].t[0] = _mm_mul_ps(half,_mm_add_ps(p[2].t[0],p[3].t[0]));
 patch1.p[2].t[1] = _mm_mul_ps(half,_mm_add_ps(p[2].t[1],p[3].t[1]));
 patch1.p[2].t[2] = _mm_mul_ps(half,_mm_add_ps(p[2].t[2],p[3].t[2]));
 const sse_t p11x = _mm_mul_ps(half,_mm_add_ps(p[1].t[0],p[2].t[0]));
 patch0.p[2].t[0] = _mm_mul_ps(half,_mm_add_ps(patch0.p[1].t[0],p11x));
 patch1.p[1].t[0] = _mm_mul_ps(half,_mm_add_ps(patch1.p[2].t[0],p11x));
 const sse_t p11y = _mm_mul_ps(half,_mm_add_ps(p[1].t[1],p[2].t[1]));
 patch0.p[2].t[1] = _mm_mul_ps(half,_mm_add_ps(patch0.p[1].t[1],p11y));
 patch1.p[1].t[1] = _mm_mul_ps(half,_mm_add_ps(patch1.p[2].t[1],p11y));
 const sse_t p11z = _mm_mul_ps(half,_mm_add_ps(p[1].t[2],p[2].t[2]));
 patch0.p[2].t[2] = _mm_mul_ps(half,_mm_add_ps(patch0.p[1].t[2],p11z));
 patch1.p[1].t[2] = _mm_mul_ps(half,_mm_add_ps(patch1.p[2].t[2],p11z));
 const sse_t p30x = _mm_mul_ps(half,_mm_add_ps(patch0.p[2].t[0],
                                               patch1.p[1].t[0]));
 patch0.p[3].t[0] = p30x;
 patch1.p[0].t[0] = p30x;
 const sse_t p30y = _mm_mul_ps(half,_mm_add_ps(patch0.p[2].t[1],
                                               patch1.p[1].t[1]));
 patch0.p[3].t[1] = p30y;
 patch1.p[0].t[1] = p30y;
 const sse_t p30z = _mm_mul_ps(half,_mm_add_ps(patch0.p[2].t[2],
                                               patch1.p[1].t[2]));
 patch0.p[3].t[2] = p30z;
 patch1.p[0].t[2] = p30z;
}
```

*Figure 6.7: Vertical patch refinement by applying the de Casteljau algorithm. The original patch is subdivided into two sub-patches. Note that the algorithm only relies on multiply-add sequences.*

### 6.3.2 Refinement Implementation

The refinement operation itself is based on the de Casteljau algorithm [Foley97], which splits the patch in half along a chosen parametric direction (either 'v' or 'u') producing two refined child patches. Alternating the parametric direction at each refinement level ensures uniform refinement.

Refinement in the 'v' direction operates mainly vertically, following the most advantageous way. The operations performed for the de Casteljau algorithm are simple affine combinations of control points. The algorithm can thus be very efficiently implemented by using only 18 SSE additions and 18 SSE multiplications, as shown in Figure 6.7.

Due to the restrictions of SSE in horizontal operations, refinement in the 'u' direction is slightly more costly. Additional 'swizzling' operations need to be inserted to operate horizontally.

### 6.3.3 Final Intersection Test

As soon as the final refinement level is reached a final intersection step is executed. Instead of considering the patch as a triangle mesh and performing a sequence of ray-triangle intersection tests [Benthin04], a reduction of the bicubic Bézier patch to a bilinear representation is applied and the intersection point is computed analytically. The eventual loss of accuracy due to the reduction is acceptable if a sufficiently large number of refinement steps has been applied before.

The four corner control points of the bicubic Bézier patch are set as the four (possibly non-coplanar) points defining the *bilinear patch*. Using a simplified bilinear representation largely reduces the complexity of the intersection algorithm. An excellent survey and source code for computing bilinear patch intersections can be found in [Ramsey04].

| Step | CPU Cycles |
|---|---|
| Pruning | 86 |
| Refinement(u) | 244 |
| Refinement(v) | 94 |
| Final Intersection | 280 |

*Table 6.1: Number of average CPU cycles for each of the core operations for intersection based on uniform refinement. All core operations cost less than* 300 *cycles. Note that for refinement in the 'u' direction, additional shuffle operations are required, resulting in higher costs than for refinement in 'v'.*

### 6.3.4 Performance of Core Operations

By implementing the recursive algorithm through an iterative sequence, additional function call overhead is avoided. As the maximum refinement level is

known in advance, the memory for storing refined patches is constant and can therefore be allocated in advance (avoiding costly memory allocation calls). As an example, the ray-patch intersection code reserves $32 * 192 = 6144$ bytes for a maximum refinement level of 32.

Table 6.1 shows with careful optimization, almost all core operations can be executed in less than 300 cycles (measured using the internal CPU clock cycle counter). Note that the current implementation still leaves room for optimization, allowing for even faster execution.

### 6.3.5 Extension for Ray Bundles

The uniform refinement approach is especially suitable for tracing ray bundles because the refinement operations are independent of the number of rays within a bundle. This allows for efficiently amortizing the cost of refinement over the rays within the bundle. Nevertheless, the other core operations for tracing bundles need to be adjusted accordingly.

Pruning for ray bundles has to be adapted in the following way: If even one ray in the bundle indicates an intersection of a patch, it must not be pruned. As long as coherence within the ray bundle is high enough almost all rays will produce the same decision in the pruning step.

As long as all rays can be grouped within a coherent beam (e.g. for primary rays), the pruning test can be simplified further. By constructing four planes out of the four corner rays spanning the beam one can easily decide (distance to plane) if all control points are outside the beam or not. Using this technique, the number of required plane tests can be significantly reduced. Instead of constructing two planes for each ray, only four planes for the entire bundle are needed. The complete pruning operation based on the four corner rays has an average cost of 240 cycles per 16-ray bundle, corresponding to an average of only 15 cycles per ray. The code in Figure 6.6 can be easily adapted to test against four instead of two planes.

The final intersection step must be applied for all rays within a bundle and cannot be easily amortized. It therefore turns out that this step is very time-critical. As SSE operates on four data elements at once it is most beneficial to perform the final intersection in bundles of four rays. Bilinear patch intersection as presented in [Ramsey04] can be coded easily using SSE for four rays in parallel. This parallel bilinear intersection code is able to reduce the average cost per ray for the final intersection step from 280 to only 88 cycles.

## 6.4 Newton Iteration

Newton iteration is a common technique for solving equations of the form $f(x) = 0$. It approximates the roots of function $f(x)$ by using the first terms of the *Taylor series* for $f(x)$. The Taylor series of $f(x)$ regarding point $x = x_0 + \epsilon$ is defined as

$$f(x_0 + \epsilon) = f(x_0) + f^{'}(x_0)\epsilon + \frac{1}{2}f^{''}(x_0)\epsilon^2 + ... \tag{6.6}$$

A restriction to only the first derivative results in

$$f(x_0 + \epsilon) \approx f(x_0) + f^{'}(x_0)\epsilon \tag{6.7}$$

By starting from an initial guess $x_0$, Equation 6.7 can be used to determine the value for $\epsilon$ by setting $\epsilon_0 = \epsilon$

$$\epsilon_0 = -\frac{f(x_0)}{f^{'}(x_0)} \tag{6.8}$$

The resulting $\epsilon_0$ can be seen as the offset to the root's position. Through continuous adjustment of

$$\epsilon_n = -\frac{f(x_n)}{f^{'}(x_n)} \tag{6.9}$$

an iterative sequence

$$x_{n+1} = x_n - \frac{f(x_n)}{f^{'}(x_n)} \tag{6.10}$$

is defined. The sequence converges to the root if the starting value $x_0$ is sufficiently close to this value. The need to chose a sufficiently close starting value is at the same time a major disadvantage of this method because of the instability near a horizontal asymptote or a local extremum. If the Newton iteration converges, it does so quadratically, which intuitively means that the number of correct digits of the root approximation roughly doubles in every step.

In the following sections, techniques for applying the Newton iteration in order to solve the ray-patch intersection problem will be discussed. Section 6.4.1 will illustrate the algorithm itself, while Section 6.4.2 to Section 6.4.4 will provide guidelines for an SSE implementation and will discuss an extension for supporting ray bundles. Section 6.4.5 will present results and will discuss limitations and bottlenecks.

### 6.4.1 Newton Iteration Algorithm

As shown by Martin et al. [Martin00], the Newton iteration represents an efficient framework for handling the ray-patch intersection problem. Starting from two perpendicular planes

$$\begin{aligned} P_0 &:= (N_0, d_0) \quad N_0\, x + d_0 = 0 \\ P_1 &:= (N_1, d_1) \quad N_1\, x + d_1 = 0 \qquad N_i \in \mathbb{R}^3,\ d_i \in \mathbb{R} \end{aligned}$$

whose intersection represents the ray, the aim is to find the roots for the following function

$$F(u,v) = \begin{pmatrix} N_0\, P(u,v) + d_0 \\ N_1\, P(u,v) + d_1 \end{pmatrix} \tag{6.11}$$

The Newton iteration for this case looks as follows

$$\begin{pmatrix} u_{n+1} \\ v_{n+1} \end{pmatrix} = \begin{pmatrix} u_n \\ v_n \end{pmatrix} - P(u_n, v_n)\, J^{-1}(u_n, v_n) \tag{6.12}$$

The term $J$ is the *Jacobian* matrix

$$J = \begin{pmatrix} N_0 \frac{\partial}{\partial u} P(u,v) & N_0 \frac{\partial}{\partial v} P(u,v) \\ N_1 \frac{\partial}{\partial u} P(u,v) & N_1 \frac{\partial}{\partial v} P(u,v) \end{pmatrix} \tag{6.13}$$

where $J^{-1}$ is defined as

$$J^{-1} = \frac{adj(J)}{det(J)} \tag{6.14}$$

As $J$ is a $2 \times 2$ matrix, the adjoint of J is

$$adj(J) = \begin{pmatrix} J_{22} & -J_{12} \\ -J_{21} & J_{11} \end{pmatrix} \tag{6.15}$$

The Newton iteration stops as soon as

$$||P(u_n, v_n)|| < \epsilon\,, \tag{6.16}$$

where $||P(u,v)||$ defines the distance of $P(u,v)$ to the ray (more precisely, to the two ray planes) and $\epsilon$ is defined as an accuracy threshold. As an early exit point, Martin et al. [Martin00] suggested the following condition:

$$||P(u_{n+1}, v_{n+1})|| > ||P(u_n, v_n)|| \tag{6.17}$$

If for the current iteration the distance to the ray is greater than the distance of the previous iteration, an early exit is performed. Additionally, the iteration stops if the number of iterations is greater than a predefined threshold.

If the Newton iteration returns a valid solution $(u_*, v_*)$, the distance of the corresponding intersection point needs to be compared with the ray origin. $P(u, v)$ is therefore evaluated at the parameter values $(u_*, v_*)$ in order to obtain the intersection point. By projecting the intersection point along the ray direction, the distance $t$ can be computed as

$$t = ray.direction * (P(u, v) - ray.origin) \tag{6.18}$$

Equations 6.13 and 6.16 demonstrate that the actual implementation requires only two core operations:

- Evaluation of the partial derivatives $\frac{\partial}{\partial u}P(u, v)$ and $\frac{\partial}{\partial v}P(u, v)$.
- Evaluation of $P(u, v)$.

As evaluating $P(u, v)$ is very similar to the evaluation of $\frac{\partial}{\partial u}P(u, v)$, the focus will be on optimizing a code sequence which handles both cases (see Section 6.4.2).

Additionally, the Newton iteration algorithm allows for performing the intersection computation for multiple rays in parallel, which is beneficial for handling ray bundles. As with triangle intersection (see Chapter 5), it is beneficial to perform the intersection for patches for four rays in parallel (see Section 6.4.2).

### 6.4.2 SSE Implementation

Geimer et al. [Geimer05] showed that the evaluation of $P(u, v)$ and the partial derivatives can be efficiently performed by computing a vector-matrix-vector product. The modified control point matrix $P^*_{ij}$ of Section 6.1.2 avoids the required evaluation of the *Bernstein* basis functions (see Equation 6.4). Therefore, $P(u, v)$ can be evaluated by the following vector-matrix-vector product:

```
inline R3 Evaluate(const BicubicBezierPatch3D &patch,
                   const float u,
                   const float v)
{
  const sse_t V = _mm_setr_ps(v*v*v,v*v,v,1.0f);
  const sse_t u2 = _mm_set_ps1(u);
  const sse_t u1 = _mm_mul_ps(u2,u2);
  const sse_t u0 = _mm_mul_ps(u1,u2);

  const sse_t X = _mm_add_ps(_mm_add_ps(_mm_mul_ps(patch.p[0].t[0],u0),
                                        _mm_mul_ps(patch.p[1].t[0],u1)),
                             _mm_add_ps(_mm_mul_ps(patch.p[2].t[0],u2),
                                        patch.p[3].t[0]));
  const float x = _mm_cvtss_f32(sseHorizontalAdd(_mm_mul_ps(X,V)));

  const sse_t Y = _mm_add_ps(_mm_add_ps(_mm_mul_ps(patch.p[0].t[1],u0),
                                        _mm_mul_ps(patch.p[1].t[1],u1)),
                             _mm_add_ps(_mm_mul_ps(patch.p[2].t[1],u2),
                                        patch.p[3].t[1]));
  const float y = _mm_cvtss_f32(sseHorizontalAdd(_mm_mul_ps(Y,V)));

  const sse_t Z = _mm_add_ps(_mm_add_ps(_mm_mul_ps(patch.p[0].t[2],u0),
                                        _mm_mul_ps(patch.p[1].t[2],u1)),
                             _mm_add_ps(_mm_mul_ps(patch.p[2].t[2],u2),
                                        patch.p[3].t[2]));
  const float z = _mm_cvtss_f32(sseHorizontalAdd(_mm_mul_ps(Z,V)));
  return R3(x,y,z);
}
```

*Figure 6.8: Evaluating $P(u, v)$ by performing a vector-matrix-vector product by SSE instructions. Through the SOA layout, horizontal operations are required to compute the sum of all register elements.*

$$P(u,v) = \begin{bmatrix} u^3 \\ u^2 \\ u \\ 1 \end{bmatrix}^T \begin{bmatrix} P^*_{00} & P^*_{01} & P^*_{02} & P^*_{03} \\ P^*_{10} & P^*_{11} & P^*_{12} & P^*_{13} \\ P^*_{20} & P^*_{21} & P^*_{22} & P^*_{23} \\ P^*_{30} & P^*_{31} & P^*_{32} & P^*_{33} \end{bmatrix} \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix} \tag{6.19}$$

Additionally, the evaluation of $P(u, v)$ can be efficiently implemented using SSE instructions. The implementation shown in Figure 6.8 starts with $u$ and $v$ as float values. Shuffle operations are required to copy float values into all register elements. Moreover, adding elements horizontally within a register requires additional operations.

Computing the derivatives in the same way allows for reducing the number of instructions because components which are multiplied by zero are not considered any further. Table 6.2 shows the average number of cycles for three different core operations. Even though almost all core operations require less than 100 cycles, the cost could be further reduced as soon as processor architectures offer better support for horizontal and shuffle operations.

| Step | Cycles |
|---|---|
| Evaluate | 90 |
| Derivative(u) | 90 |
| Derivative(v) | 79 |

*Table 6.2: Average number of CPU cycles for each of the core operations (single ray). All core operations require less than* 100 *cycles. For the derivative in the 'u' direction, fewer shuffle operations are required, resulting in lower costs than for the derivative in 'v'.*

### 6.4.3 Extension for Ray Bundles

Adapting the intersection for ray bundles is straightforward. As SSE allows for operating on four data elements using a single instruction, it is beneficial to apply the Newton iteration for bundles of four rays in parallel. Performing four Newton iterations in parallel makes it necessary to evaluate $P(u, v)$ for two vectors $U = (u_0, u_1, u_2, u_3)$ and $V = (v_0, v_1, v_2, v_3)$ in parallel (see Figure 6.9). Even though the layout of $U$ and $V$ avoids additional horizontal operations (such as required for the single ray implementation), the four-ray implementation requires that every control point coordinate be copied into the four register elements by an additional shuffle instruction.

One could avoid the shuffle instruction by storing every control point coordinate as an SSE data type, but this would increase the patch data storage costs by a factor of 4, thus significantly affecting the advantage of compact data representation. On the other hand, one could use larger bundles (of more than four rays) to amortize the cost for the shuffle instruction, but the limited number of registers makes this approach inefficient.

Table 6.3 shows the average number of required CPU cycles. On a per ray basis, the costs for the core operations are reduced, in particular for evaluating the derivatives.

```
inline void Evaluate4(const sse_t u,
                      const sse_t v,
                      const BicubicBezierPatch3D &patch,
                      SSEVec4 &result)
{
  const sse_t u2 = u;
  const sse_t u1 = _mm_mul_ps(u,u);
  const sse_t u0 = _mm_mul_ps(u,u1);
  const sse_t v2 = v;
  const sse_t v1 = _mm_mul_ps(v,v);
  const sse_t v0 = _mm_mul_ps(v,v1);
  // == X ==
  const sse_t t0 = patch.p[0].t[0];
  const sse_t t1 = patch.p[1].t[0];
  const sse_t t2 = patch.p[2].t[0];
  const sse_t t3 = patch.p[3].t[0];
  const sse_t x0 = _mm_add_ps(_mm_add_ps(_mm_mul_ps(SHUFFLE(t0,0),u0),
                                         _mm_mul_ps(SHUFFLE(t1,0),u1)),
                              _mm_add_ps(_mm_mul_ps(SHUFFLE(t2,0),u2),
                                         SHUFFLE(t3,0)));
  const sse_t x1 = _mm_add_ps(_mm_add_ps(_mm_mul_ps(SHUFFLE(t0,1),u0),
                                         _mm_mul_ps(SHUFFLE(t1,1),u1)),
                              _mm_add_ps(_mm_mul_ps(SHUFFLE(t2,1),u2),
                                         SHUFFLE(t3,1)));
  const sse_t x2 = _mm_add_ps(_mm_add_ps(_mm_mul_ps(SHUFFLE(t0,2),u0),
                                         _mm_mul_ps(SHUFFLE(t1,2),u1)),
                              _mm_add_ps(_mm_mul_ps(SHUFFLE(t2,2),u2),
                                         SHUFFLE(t3,2)));
  const sse_t x3 = _mm_add_ps(_mm_add_ps(_mm_mul_ps(SHUFFLE(t0,3),u0),
                                         _mm_mul_ps(SHUFFLE(t1,3),u1)),
                              _mm_add_ps(_mm_mul_ps(SHUFFLE(t2,3),u2),
                                         SHUFFLE(t3,3)));
  result.t[0] = _mm_add_ps(_mm_add_ps(_mm_mul_ps(x0,v0),
                                      _mm_mul_ps(x1,v1)),
                           _mm_add_ps(_mm_mul_ps(x2,v2),
                           x3));
  // == Y,Z similar ==
  // ..
}
```

*Figure 6.9: Evaluating $P(U,V)$ for two vectors $U = (u_0, u_1, u_2, u_3)$ and $V = (v_0, v_1, v_2, v_3)$ by performing a vector-matrix-vector product using SSE instructions. No horizontal operations are required but many shuffle instructions, because each control point coordinate has to be copied into the four register elements.*

| Step | Cycles / Per Four-Ray Bundle | Cycles / Per Ray |
|---|---|---|
| Evaluate | 350 | 87.5 |
| Derivative(u) | 255 | 63.75 |
| Derivative(v) | 244 | 61 |

*Table 6.3: Average number of CPU cycles for each of the core operations when operating on four rays in parallel. Computing derivatives requires fewer operations, resulting in lower costs.*

### 6.4.4 Newton Iteration Convergence

The greatest difficulty by the Newton iteration is the fact that convergence depends on the starting value. If the starting point is chosen too far away from the correct solution, the Newton iteration can converge to a wrong solution (get "stuck" in a local minimum), converge slowly, or even fail to converge at all.

In order to obtain a good starting value, researchers [Martin00] have suggested to subdivide a given patch into a set of sub-patches (based on a flatness criteria [Guthe02]) and to build a spatial index structure from these sub-patches. Each sub-patch corresponds to a specific parametric domain of the original patch. A ray, or a ray bundle, is then traversed through this spatial index structure and, in the case of sub-patch intersection, the starting value of the Newton iteration is initialized to the center of the corresponding parametric domain of the sub-patch.

If the patch is sufficiently flat, patch subdivision for initializing starting values works quite well. Nevertheless, it can never guarantee convergence. Note that the creation of sub-patches typically requires a chosen flatness or curvature threshold, which has to be manually adjusted depending on the curvature of patches.

### 6.4.5 Discussion

The convergence speed of a Newton iteration-based intersection algorithm depends mainly on the distance between the starting value and the final result. If the distance is sufficiently small, Newton iteration offers quadratic convergence, which makes it extremely fast. Nevertheless, the standard approach of determining a sufficiently close starting value by traversing an additional spatial index structure (see Section 6.4.4), which is constructed per patch, has serious drawbacks. First of all, the overall memory requirements are higher because of the additional spatial structure per patch, making rendering of

highly complex scenes difficult. Furthermore, the larger memory footprint increases the probability of cache misses. Finally, even a finely resolved spatial index structure does not guarantee convergence in all potential cases.

The main advantage of a Newton iteration-based approach is that the ray-patch intersection costs essentially depend on the costs for evaluating $P(U, V)$ and the corresponding derivatives. A closer look at the code itself shows that the evaluation almost exclusively consists of multiply-add sequences. This is similar to the implementation of core operations for uniform refinement. And here too, the performance could be significantly increased using an exclusive multiply-add SSE instruction and more SSE registers for storing temporary values.

## 6.5 Newton Iteration and Krawczyk Operator

As discussed in Section 6.4, a Newton iteration-based algorithm can fail to converge if the starting point is chosen too far away from the correct solution. Fortunately, the *Krawczyk operator* [Krawczyk69, Krawczyk70] is able to determine for a chosen starting point if the Newton iteration will converge or not. Therefore, the Krawczyk operator can be used as an extension to the standard Newton iteration-based intersection in order to identify safe starting regions. The implementation described in this section is an extension of the original work by Toth et al. [Toth85].

Before explaining the Krawczyk operator and the extension of the Newton iteration-based intersection in detail, the fundamentals of *interval arithmetic*, which form the basis of a Krawczyk operator, are presented first.

### 6.5.1 Interval Arithmetic

An *interval* (over $\mathbb{R}$) is defined as a bounded set of real numbers $X = [X_{min}, X_{max}] = \{x \in \mathbb{R} : X_{min} \leq x \leq X_{max}\}$. For intervals, which in the following will be indicated in capital letters, the standard arithmetic operations $\{+, -, *, /\}$ are defined as:

$$X + Y = [X_{min} + Y_{min}, X_{max} + Y_{max}] \tag{6.20}$$

$$X - Y = [X_{min} - Y_{max}, X_{max} - Y_{min}] \tag{6.21}$$

$$X * Y = [min_{x*y}, max_{x*y}] \tag{6.22}$$

$$1.0/X = [1.0/X_{max}, 1.0/X_{min}], 0 \notin X \tag{6.23}$$

When multiplying two intervals, the upper, respectively lower bound of the result is computed by determining the maximum resp. minimum of the four

combinations: $min_{x*y} = min(X_{min}Y_{min}, X_{min}Y_{max}, X_{max}Y_{min}, X_{max}Y_{max})$ and $max_{x*y} = max(X_{min}Y_{min}, X_{min}Y_{max}, X_{max}Y_{min}, X_{max}Y_{max})$.

Additionally, operations between an interval $X$ and a scalar value $y \in \mathbb{R}$ are defined as:

$$X + y = [X_{min} + y, X_{max} + y] \tag{6.24}$$
$$X * y = [min(X_{min} * y, X_{max} * y), max(X_{min} * y, X_{max} * y)] \tag{6.25}$$

All operations are associative and commutative. Note that distributivity does not hold true, only sub-distributivity:

$$X * (Y + Z) \subset X * Y + X * Z \tag{6.26}$$

For evaluating terms, more accurate results can be achieved by factorizing the elements as much as possible.

To implement the Krawczyk operator, two additional operators are required: The *midpoint* operator $m(X)$ and the *width* operator $w(X)$. These are defined as

$$m(X) = \frac{X_{min} + X_{max}}{2} \tag{6.27}$$
$$w(X) = X_{max} - X_{min} \tag{6.28}$$

A natural extension to intervals are *interval vectors*. An $m$-dimensional *interval vector* is defined as $\mathbb{X} = (X_0, .., X_{m-1}) \in \mathbb{R}^m$, where $X_i$ is an interval over $\mathbb{R}$.

For two interval vectors $\mathbb{X}$ and $\mathbb{Y} \in \mathbb{R}^m$, the operations $\Theta \in \{+, -, *, /\}$ are defined componentwise

$$\mathbb{X}\Theta\mathbb{Y} = (X_0\Theta Y_0, .., X_{m-1}\Theta Y_{m-1}) \tag{6.29}$$

Similarly, the midpoint and the width operator for interval vectors are also defined componentwise

$$m(\mathbb{X}) = (m(X_0), .., m(X_{m-1})) \tag{6.30}$$
$$w(\mathbb{X}) = (w(X_0), .., w(X_{m-1})) \tag{6.31}$$

In the same manner as for defining functions over $\mathbb{R}$, one can extend the functions to intervals over $\mathbb{R}$. An *interval extension* of a function

$$f(x_0, .., x_{m-1}) = (y_0, .., y_{n-1}), x_i, y_j \in \mathbb{R} \tag{6.32}$$

is defined as

$$F(X_0, .., X_{m-1}) = (Y_0, .., Y_{n-1}) \tag{6.33}$$

where $X_i, Y_j$ are intervals over $\mathbb{R}$.

### 6.5.2 Krawczyk Operator

The Krawczyk operator $K(X_i, y, Y)$ is defined as:

$$\begin{aligned} K(X_i, y, Y) &= y - Y * f(y) + (|Y| * A * e * [-1, 1]) && (6.34)\\ y &= m(X_i) && (6.35)\\ Y &= m(F'(X_i))^{-1} && (6.36)\\ A_{ij} &= \frac{1}{2}(w(F'(X_i)))_{ij} && (6.37)\\ e_i &= \frac{1}{2}(w(X_i))_i && (6.38) \end{aligned}$$

where $X_i$ is an interval extension of $x_i \in \mathbb{R}$, $y \in X_i$, and $Y$ a non-singular matrix. Without loss of generality, $y = m(X_i)$ and $Y = m(F'(X_i))^{-1}$. The fundamental idea of applying the Krawczyk operator is to determine safe convergence regions within interval $X_i$.

From the given interval $X_i$, the Krawczyk operator $K(X_i, y, Y)$ computes a new interval $X_i'$. Based on the new interval $X_i'$, a convergence criteria is evaluated that determines if $X_i$ can be considered as a safe region for starting a Newton iteration.

For a detailed description of the mathematical background, see [Krawczyk69, Krawczyk70, Toth85].

### 6.5.3 Krawczyk-Moore Intersection Algorithm

Based on the work by Moore [Moore77], Toth [Toth85] proposed an algorithm that uses the Krawczyk operator to identify safe regions for the Newton iteration. If a solution for function $f(x) = 0$ with $x \in X_0$ exists, the Krawczyk-Moore test will return an interval $X_{safe}$ for which all $x \in X_{safe}$ converge to a non-ambiguous solution. The underlying process can be formulated as an iterative algorithm starting with the initial interval $X_0$. The intervals $X_i$ builds a sequence of nested intervals:

$$X_1 = X_0 \bigcap K(X_0, m(X_0), m(F'(X_0))^{-1}) \quad (6.39)$$

$$X_2 = X_1 \bigcap K(X_1, m(X_1), m(F'(X_1))^{-1}) \quad (6.40)$$

$$... \quad (6.41)$$

$$X_{i+1} = X_i \bigcap K(X_i, m(X_i), m(F'(X_i))^{-1}) \quad (6.42)$$

The sequence can be terminated if one of the follow conditions holds true:

- $X_i$ is a safe region for starting the Newton iteration
- $X_i = \varnothing$: There exists no valid solution for $X_i$
- $w(X_i) < Epsilon$ and $w(F'(X_i)) < Epsilon$: $F(X_i)$ can be considered as almost planar. In this case only a single Newton step using $m(X_i)$ has to be performed.
- $X_i = X_{i-1}$: since the intervals are not shrinking, subdivide $X_i$ in half and continue recursively with each sub-interval.

What remains now is to define the conditions when $X_i$ can be considered a safe starting region. If either

$$K(X_i, m(X_i), m(F'(X_i))^{-1}) \subset X_i \quad (6.43)$$

$$r = ||I - m(F'(X_i))^{-1} * F'(X_i)|| < 1 \quad (6.44)$$

or

$$||m(X_i) - n(m(X_i))|| < (1 - r) * a \quad (6.45)$$

$$r = ||I - m(F'(X_i))^{-1} * F'(X_i)|| < 1 \quad (6.46)$$

with $a = min_i(\frac{1}{2}w(X_i))$ and $B(X_i, a) = x \in X_i : ||x - m(X_i)|| \leq a$ is fulfilled, the Newton iteration will converge for all $x \in X_i$, respectively $x \in B(X_i, a)$ [Toth85].

As convergence of the interval sequence $X_i$ mainly depends on the interval extension of the derivatives $S_U(X_i)$ and $S_V(X_i)$, it is important to compute these (3-dimensional) intervals as accurately as possible. The approach of computing, for example, $S_U(X_i)$ by

$$S_V(U,V) = \begin{bmatrix} V^3 \\ V^2 \\ V \\ 1 \end{bmatrix}^T \begin{bmatrix} P^*_{00} & P^*_{01} & P^*_{02} & P^*_{03} \\ P^*_{10} & P^*_{11} & P^*_{12} & P^*_{13} \\ P^*_{20} & P^*_{21} & P^*_{22} & P^*_{23} \\ P^*_{30} & P^*_{31} & P^*_{32} & P^*_{33} \end{bmatrix} \begin{bmatrix} 3U^2 \\ 2U \\ U \\ 0 \end{bmatrix} \tag{6.47}$$

where $U$ and $V$ are interval extensions and $P^*_{ij}$ is the modified control point matrix, is fast and efficient. Given interval distributivity, however, it does not ensure optimal accuracy (see Equation 6.26). The limited accuracy increases the number of iterations of the Krawczyk-Moore test to identify a safe starting region. Therefore, it is more beneficial to compute derivatives the following way: Given the original patch with its control points $P_{ij}$ (not multiplied with the Bernstein matrix), a new patch with control points $P^R_{ij}$ is computed that corresponds to the original patch restricted to interval $X_i$. From the new patch, a reduced $3 \times 4$ control point matrix $Q_{ij}$ is created. Note that for $S_U$

$$Q_{ij} = \frac{3}{w(U)}(P^R_{ij+1} - P^R_{ij}) \tag{6.48}$$

and for $S_V$

$$Q_{ij} = \frac{3}{w(V)}(P^R_{i+1j} - P^R_{ij}) \tag{6.49}$$

The interval extension of derivatives $S_U(X_i)$ and $S_V(X_i)$ can be easily obtained by determining the minimum and maximum intervals over the components of the reduced control point matrix $Q_{ij}$.

Note that this approach requires additional clipping operations due to the restrictions of patch control points to the interval $X_i$. Even though the clipping operations can be efficiently performed by de Casteljau subdivisions, the four subdivision steps are costly. However, the interval extension obtained from the clipping step is typically more accurate than the result of the matrix evaluation approach. The higher accuracy directly translates to a three to four times smaller number of iterations for the Krawczyk-Moore test (see Section 6.10).

### 6.5.4 Implementation

The implementation of an intersection algorithm based on the Krawczyk operator can be directly combined with the Newton iteration algorithm from

Section 6.4. Finding a safe starting region is just a preamble for the actual Newton iteration step. Therefore, the main task is to efficiently implement the evaluation of the Krawczyk operator.

In order to provide an easy and encapsulated access to interval arithmetic, a few simple C++ classes are used (see Figure 6.10).

```
class Interval {
public:
  float min;
  float max;
  // ... arithmetic operators for a one-dimensional interval vector
};
class Interval3D {
public:
  Interval interval[3];
  // ... arithmetic operators for a three-dimensional interval vector
};
class IntervalMatrix2x2 {
public:
  Interval interval[2][2];
  // ... arithmetic operators for a 2x2-dimensional interval matrix
};
```

*Figure 6.10: Simple C++ classes for implementing interval arithmetic.*

Based on the interval arithmetic classes of Figure 6.10, a brief description of each step required to compute the Krawczyk operator will be given.

**$\mathbf{S_U(X_i)}$ and $\mathbf{S_V(X_i)}$:** The partial derivatives of the patch with respect to interval $X_i$ can be computed either by performing an interval-vector matrix interval-vector product or by determining the minimum and maximum intervals of the reduced control point matrix. In either case, the result is an *Interval3D* class, corresponding to a three-dimensional interval vector.

**$\mathbf{F_U(X_i)}$ and $\mathbf{F_V(X_i)}$:** These are the partial derivatives of $F(X_i)$ which can be efficiently computed by representing the ray as the intersection of two planes $(N_U, d_U)$ and $(N_V, d_V)$, where $N_U, N_V \in \mathbb{R}^3$ and $d_U, d_V \in \mathbb{R}$. In this case, the computation of $F_U(X_i)$ and $F_V(X_i)$ only involves four dot products:

$$F_U(X_i) = \begin{pmatrix} N_U * S_U(X_i) \\ N_V * S_U(X_i) \end{pmatrix}, F_V(X_i) = \begin{pmatrix} N_U * S_V(X_i) \\ N_V * S_V(X_i) \end{pmatrix} \quad (6.50)$$

The result of, e.g., $N_U * S_U(X_i)$ is a one-dimensional interval. Therefore, the result of $F_U(X_i)$ and $F_V(X_i)$ is stored into an *IntervalMatrix2x2* class, which is a $2 \times 2$ interval matrix.

$\mathbf{Y = m(F'(X_i))^{-1}}$**:** As $F_U(X_i)$ and $F_V(X_i)$ are partial derivatives of $F(X_i)$, the derivative of $F(X_i)$ is the Jacobian matrix $J = (F_U(X_i)F_V(X_i))$. Before computing the inverse of the $2 \times 2$ Jacobian interval matrix $J$, the midpoint operation $m(X_i)$ is applied first. This transfers the $2 \times 2$ interval matrix into a real-valued $2 \times 2$ matrix. The inverse is computed by $J^{-1} = \frac{adj(J)}{detJ}$.

$\mathbf{A_{ij} = \frac{1}{2}(w(F'(X_i)))_{ij}}$**:** Performing the *width* operation transforms the previously computed $2 \times 2$ interval matrix into the real-valued $2 \times 2$ matrix $A$.

$\mathbf{e_i = \frac{1}{2}(w(X_i))_i}$**:** Computing $e_i$ only requires the *width* operator applied on the current interval $X_i$.

$\mathbf{f(y)}$**:** The function $f(y) = f(m(X_i))$ can be quickly evaluated by performing a vector-matrix-vector product (see Section 6.4).

All these steps are required for an implementation of an intersection algorithm using the Krawczyk operator and Newton iteration. Figure 6.11 shows a pseudo C++ code for an example implementation.

In a first step, the algorithm performs a patch clipping operation (starting from the original patch) based on the current parametric interval *UV*. For implementing the *Clip* operation, the patch refinement code from Section 6.3 can be reused. Note that the clipping operation is performed in every iteration of the algorithm and should therefore be as optimized as possible.

In order to avoid unnecessary iterations of the Krawczyk-Moore test, an early pruning test is performed based on the axis-aligned bounding box of the clipped patch and the ray. If the ray intersects the bounding box, the partial derivatives (*Su* and *Sv*) of the clipped patch are computed and the results are stored in two three-dimensional interval vectors (*su* and *sv*). Figure 6.12 shows an example implementation of *Sv* using SSE instructions.

The $2 \times 2$ interval matrix *mat* of the partial derivatives is computed by performing four dot products between the patch derivatives *su* and *sv* and the two ray plane equations *(Nu,du)* and *(Nv,dv)*. This interval matrix forms the basis for obtaining the two $2 \times 2$ real-valued matrices $Y$ and $A$.

After evaluating the patch at the center of the current interval *UV*, the distances *fy* of the corresponding point to the two ray planes are computed. The interval *UV*, the distances *fy*, and the matrices $Y$ and $A$ then allow for evaluating the Krawczyk operator (see Equation 6.34). If interval $k$ returned

```
BicubicBezierPatch3D patch; Ray ray; // current patch and ray data
R3 Nu,Nv; float du,dv; // two ray planes (Nu,du) and (Nv,dv)
Interval2D stack[MAX_STACK_DEPTH];
Interval2D *sptr = stack;
*sptr++ = Interval2D(0.0f,0.0f,1.0f,1.0f);
while(sptr != stack) {
  sptr--; Interval2D UV = *sptr; // get interval from stack
  while(1) {
    // -- clipped original patch based on current UV interval --
    BicubicBezierPatchSSE clipped_patch = Clip(UV,patch);
    // -- test if ray intersects bounding box of clipped patch --
    Box box = clipped_patch.GetAABB();
    if (!IntersectBox(ray,box)) break;
    // -- compute interval extension of derivatives --
    Interval3D su = Su(UV,clipped_patch);
    Interval3D sv = Sv(UV,clipped_patch);
    // -- compute interval matrix for partial derivatives --
    IntervalMatrix2x2 mat = Fuv(su,sv,Nu,Nv);
    // -- compute Matrix A and Y--
    Matrix2x2 A,Y; A = ComputeA(mat); Y = ComputeY(mat);
    // -- evaluate patch at the center of UV --
    R3 p = Evaluate(patch,UV.center());
    // -- compute distances to ray planes (Nu,du) and (Nv,dv) --
    R2 fy = ComputeDistances(p,Nu,Nv,du,dv);
    // -- compute new interval using the Krawczyk operator --
    Interval2D k = ComputeKraw(UV,Y,A,fy);
    if (Disjoint(UV,k)) break; // -- UV and k disjoint -> skip UV --
    // -- compute subset of UV and k --
    Interval2D UV_k = Subset(UV,k);
    // -- if safe interval exists -> start Newton iteration --
    if (isSafeInterval(UV,k,Y,A,fy)) {
      NewtonIteration(UV_k,ray,patch); break;
    } else {
      // -- test if no reduction could be achieved --
      if (UV_k == UV) {
      // -- subdivide UV into four subsets, continue recursively --
      Subdivide(UV,sptr[0],sptr[1],sptr[2],sptr[3]); sptr+=4; break;
    }
    UV = UV_k; // -- update UV interval --
} } }
```

*Figure 6.11: Pseudo C++ Code for a ray-patch intersection algorithm using the Krawczyk-Moore test. The interval extension of the derivatives is computed by clipping the original patch according to the 'UV' interval. If a safe interval region is identified, a Newton iteration is performed.*

```
inline float hMax(const sse_t &a) {
  const sse_t ftemp = _mm_max_ps(a,_mm_movehl_ps(a, a));
  return _mm_cvtss_f32(_mm_max_ss(ftemp,_mm_shuffle_ps(ftemp,ftemp,1)));
}
inline float hMin(const sse_t &a) {
  const sse_t ftemp = _mm_min_ps(a,_mm_movehl_ps(a, a));
  return _mm_cvtss_f32(_mm_min_ss(ftemp,_mm_shuffle_ps(ftemp,ftemp,1)));
}
inline Interval3D Sv(const Interval2D &UV,
                     const BicubicBezierPatch3D &patch)
{
  Interval3D S;
  const Interval &V = UV.interval[1];
  const float width = 3.0f / width(V);
  // -- X --
  const sse_t x0 = _mm_sub_ps(patch.p[1].t[0],patch.p[0].t[0]);
  const sse_t x1 = _mm_sub_ps(patch.p[2].t[0],patch.p[1].t[0]);
  const sse_t x2 = _mm_sub_ps(patch.p[3].t[0],patch.p[2].t[0]);
  S.interval[0].min = width * hMin(_mm_min_ps(x0,_mm_min_ps(x1,x2)));
  S.interval[0].max = width * hMax(_mm_max_ps(x0,_mm_max_ps(x1,x2)));
  // -- Y --
  const sse_t y0 = _mm_sub_ps(patch.p[1].t[1],patch.p[0].t[1]);
  const sse_t y1 = _mm_sub_ps(patch.p[2].t[1],patch.p[1].t[1]);
  const sse_t y2 = _mm_sub_ps(patch.p[3].t[1],patch.p[2].t[1]);
  S.interval[1].min = width * hMin(_mm_min_ps(y0,_mm_min_ps(y1,y2)));
  S.interval[1].max = width * hMax(_mm_max_ps(y0,_mm_max_ps(y1,y2)));
  // -- Y --
  const sse_t z0 = _mm_sub_ps(patch.p[1].t[2],patch.p[0].t[2]);
  const sse_t z1 = _mm_sub_ps(patch.p[2].t[2],patch.p[1].t[2]);
  const sse_t z2 = _mm_sub_ps(patch.p[3].t[2],patch.p[2].t[2]);
  S.interval[2].min = width * hMin(_mm_min_ps(z0,_mm_min_ps(z1,z2)));
  S.interval[2].max = width * hMax(_mm_max_ps(z0,_mm_max_ps(z1,z2)));
  return S;
}
```

*Figure 6.12: After clipping the original patch to the current interval 'UV, the interval extension of patch derivatives (here in the parametric 'v' direction) can be efficiently coded using SSE instructions. In a first step, a reduced control point matrix (of the clipped patch) is created by subtracting control point $P_{i+1j}$ from $P_{ij}$. For each of the three dimensions, the corresponding interval is set to the minimum and maximum of the control point differences (multiplied by $3.0/width(V)$).*

by the Krawczyk operator and the current interval $UV$ are disjoint, the current interval $UV$ is excluded from further processing.

If one of the convergence criteria is fulfilled (*isSafeInterval*), the current interval $UV$ is considered as a safe starting region and a Newton iteration-based intersection is performed. If no safe starting region can be determined, the algorithm performs a comparison of equality between the current interval $k$ and the subset of $k$ and $UV$. If no reduction of the parametric region can be achieved, the algorithm subdivides $UV$ into four sub-intervals and continues recursively. Otherwise, $UV$ is replaced by the subset of $k$ and $UV$, and the algorithm continues. The algorithm terminates when the interval stack is empty.

### 6.5.5 Discussion

The main advantage of the Krawczyk operator is that it avoids the convergence problems of the standard Newton iteration-based intersection algorithm, providing reliable results. No additional spatial index structures that limit the parametric starting region are required. This reduces memory requirements because only the original patch data is needed.

However, identifying safe regions by evaluating the Krawczyk operator does not come for free. In each iteration, many complex operations, e.g. patch clipping or computing derivative intervals, have to be performed. Even though most operations can be efficiently implemented using SSE instructions, significantly more SSE instructions are required than for standard Newton iteration-based intersection. For many scenes, this significantly impacts performance (see Section 6.10).

The number of iterations that are required to identify a safe convergence region largely depends on the size of the partial derivative intervals. The larger the derivative intervals, the smaller the reduction of the current parametric region. Typically, the expansion of the derivative intervals becomes larger with an increased patch curvature. Therefore, it might be beneficial to subdivide highly curved patches in a preprocessing step in order to first reduce the curvature of these patches.

## 6.6 Bézier Clipping

Another approach for solving the intersection problem is the Bézier clipping algorithm. Bézier clipping was first presented by Nishita et al. [Nishita90]. The basic idea behind the Bézier clipping algorithm is to cut away regions of the patch (and therefore parametric intervals) that are known not to be intersected by the ray. Through continuous cutting, the parametric domain associated with the remaining sub-patch converges to the solution.

Note that Bézier clipping even works for rational Bézier surfaces or Bézier curves because all operations are only performed after a projection of the patch into 2D.

The original algorithm proposed by Nishita et al. [Nishita90] suffers from certain numerical instabilities because of the underlying nature of the algorithm. Campagna et al. [Campagna97] identified those critical issues and proposed a modified version, which greatly reduced the numerical problems.

As a matter of fact, Bézier clipping is a very complex algorithm, whose implementation is not trivial. Nevertheless, even this complex algorithm can be efficiently implemented using SSE instructions. In order to provide a better understanding of implementation issues, Section 6.6.1 will first discuss the original algorithm in detail. Section 6.6.2 will present a step-by-step approach to an SSE implementation. Finally, Section 6.6.3 will deal remaining problems and bottlenecks.

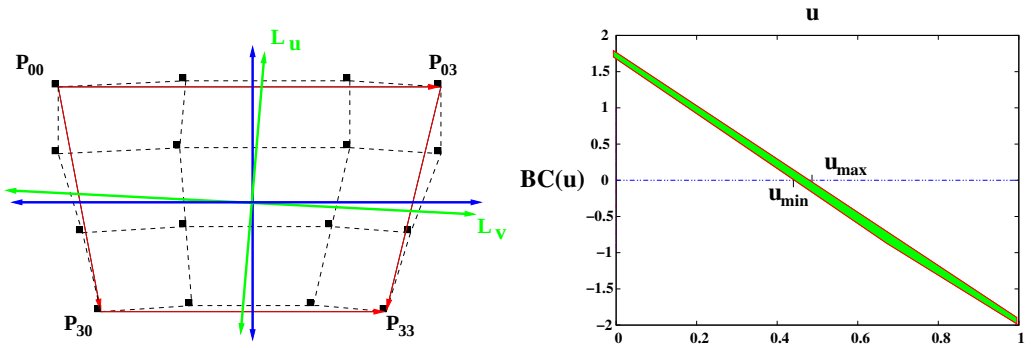



*Figure 6.13: Left Image: The distances of all 3D control points to the two ray planes (blue lines) are determined, resulting in a 2D patch representation. In order to reduce the parametric domain, two lines* $L_u = \frac{1}{2}(P_{30} - P_{00} + P_{33} - P_{03})$ *and* $L_v = \frac{1}{2}(P_{03} - P_{00} + P_{33} - P_{30})$ *are computed (green lines) which pass through the origin and are roughly perpendicular to the two parametric directions 'u' and 'v'. Right Image: The convex hull of the distances of all 2D control points to line* $L_v$*, plotted using equidistant spacing of the unit interval. The region outside of the convex hull is known to not to be intersected and can be cut away.*

### 6.6.1 The Bézier Clipping Algorithm

Bézier clipping works by cutting away regions of the patch, and thus parametric regions of the parameter domain, that are known not to be intersected

by the ray. Each ray is again represented as the intersection of two planes $pl_0(x) = N_0\, x + d_0$ and $pl_1(x) = N_1\, x + d_1$. In a first step, the distance for all control points $P_{ij}$ with respect to the two planes are computed

$$d_{ij} = (N_0\, P_{ij} + d_0, N_1\, P_{ij} + d_1) \tag{6.51}$$

This allows for performing all cutting and subdivision operations in 2D instead of 3D, thus saving 33% of the total number of operations. The distances $d_{ij}$ are used as control points for the projected 2D patch representation

$$P^{'}(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} B_i^3(u)B_j^3(v)\, d_{ij}\ ,\ d_{ij} \in \mathbb{R}^2 \tag{6.52}$$

The two lines $pl_0$ and $pl_1$ specify the coordinate axes, as shown in the left image of Figure 6.13. This reduces the intersection problem to finding values for $u$ and $v$, so that $P^{'}(u,v) = 0$. The algorithm finds potential $u$ and $v$ values by iteratively excluding parametric regions with $P^{'}(u,v) \neq 0$. Therefore, two lines $L_u$ and $L_v$ which pass through the origin are determined. Those lines should be roughly perpendicular to the two parametric directions $u$ and $v$, and are usually defined as:

$$L_u = \frac{1}{2}(P_{30} - P_{00} + P_{33} - P_{03}) \tag{6.53}$$

$$L_v = \frac{1}{2}(P_{03} - P_{00} + P_{33} - P_{30}) \tag{6.54}$$

In rare cases when either $L_u = 0$ or $L_v = 0$, the computation of $L_u$ and $L_v$ must be performed differently [Efremov05]. If $u$ is the first parametric direction to be considered, the distances $d_{ij}^{'}$ of all 2D control points to line $L_u$ are computed next. Relying on the following property of Bernstein polynomials

$$\sum_{i=0}^{n} \frac{i}{n} B_i^n(u) = u \tag{6.55}$$

an explicit Bézier curve can be defined as

$$BC(u) = \sum_{i=0}^{n} \begin{bmatrix} \frac{i}{n} \\ d_i^{'} \end{bmatrix} B_i^n(u) \tag{6.56}$$

As $BC(u)$ intersects the u-axis at some point where $P'(u,v)$ intersects $L_u$, the convex hull criteria for $BC(u)$ can be applied in order to identify regions for $u$ that do not intersect the u-axis and therefore do not intersect $L_u$ either. The right image of Figure 6.13 shows the convex hull of the points $(\frac{i}{n}, d_i')$ intersecting the u-axis at $u_{min}$ and $u_{max}$. Therefore, intervals $[0, u_{min}]$ and $[u_{max}, 1]$ can be excluded from containing potential solutions in the $u$ direction. Note that $\frac{i}{n}$ is an equidistant spacing of the unit interval.

Two de Casteljau subdivision steps are now performed in the $u$ direction in order to reduce the 2D Bézier patch $P'(u,v)$ to the reduced parametric interval $[u_{min}, u_{max}]$. The algorithm continues by alternating between $u$ and $v$ as parametric directions. Bézier clipping has three termination criteria:

- One interval is smaller than a predefined epsilon. If only one parametric direction is still greater than epsilon, the algorithm continues exclusively with the corresponding interval. If the interval for both directions is smaller than epsilon, the center of either interval is returned as a valid intersection.
- After the de Casteljau subdivision, the subdivided patch does not include the origin (with respect to the two ray planes), resulting in no intersection.
- The convex hull of the Bézier curve $BC(u)$ does not intersect the $u$-axis, resulting in no intersection.

Handling multiple intersections can be problematic because the algorithm does not need to terminate in this case. A simple solution to this problem is to subdivide the patch in half and restart the algorithm with either half provided the parametric interval reduction is less than a constant value. Nishita [Nishita90] heuristically determined this value as being 20%.

If the algorithm converges to a solution, the original patch $P(u,v)$ is evaluated at the corresponding parameter values in order to obtain the coordinates of the intersection point. By projecting the intersection point along the ray direction, distance $t$ can be computed easily by

$$t = ray.direction * (P(u,v) - ray.origin) \tag{6.57}$$

### 6.6.2 SSE Implementation

The following operations required by the Bézier clipping algorithm have been identified as suitable core operations for implementation using SSE instructions:

```
struct BicubicBezierPatch2D
{
  sse_t Nu_dist[4];
  sse_t Nv_dist[4];
};
```

*Figure 6.14: The structure stores the sixteen 2D control points of the patch $P'(u, v)$, which are the distances $d_{ij}$ of the original control points to the two ray planes, in an SSE suitable SOA format.*

- Obtaining the 2D control points $d_{ij}$ of patch $P'(u, v)$ by computing the distances of the 3D control points $P_{ij}$ to the two ray planes $pl_0$ and $pl_1$.
- Checking whether the current patch $P'(u, v)$ includes the origin.
- Computing the $L_u$ and $L_v$.
- Computing the distances $d'_{ij}$ of $d_{ij}$ to either $L_u$ or $L_v$, and determining the convex hull of the obtained distance control points. Finding $u_{min}$ and $u_{max}$ if the convex hull intersects the $u$-axis.
- Based on the reduced parametric interval $[u_{min}, u_{max}]$, subdividing the current patch two times in order to obtain sub-patch $P'(u, v)$.

In the following, the implementation for all those operations will be discussed in detail.

### Initialization 3D to 2D

Applying the Bézier clipping algorithm for bicubic Bézier patches requires a data structure for storing the 2D control points $d_{ij}$ (see Figure 6.14) for the 2D patch $P'(u, v)$. This data structure is initialized by computing the distances of the original control points to the two ray planes.

The code illustrated in Figure 6.15 shows that distances $d_{ij}$ can be efficiently computed by performing a sequence of four dot products in parallel. Additionally, a pruning test is performed by checking the signs of the computed 2D distance coordinates. If all distances have matching signs, the 2D convex hull of all control points cannot contain the origin. Thus, the ray cannot intersect the patch. Note that this additional test can be performed basically without any additional costs.

```
inline bool Init(const BicubicBezierPatch3D &patch,
                 const RayPlanesSSE &plane)
{
 unsigned int signsU = 0;
 for (unsigned int i=0;i<4;i++)
  {
   const sse_t d = _mm_add_ps(_mm_add_ps(_mm_mul_ps(plane.Nu.t[0],
                                                    patch.p[i].t[0]),
                                         _mm_mul_ps(plane.Nu.t[1],
                                                    patch.p[i].t[1])),
                              _mm_mul_ps(plane.Nu.t[2],patch.p[i].t[2]));
   const sse_t d_f = _mm_sub_ps(d,plane.du);
   Nu_dist[i] = d_f;
   signsU |= _mm_movemask_ps(d_f) << (4*i);
  }
 if (signsU == 0xffff || signsU == 0) return false;
 unsigned int signsV = 0;
 for (unsigned int i=0;i<4;i++)
  {
   const sse_t d = _mm_add_ps(_mm_add_ps(_mm_mul_ps(plane.Nv.t[0],
                                                    patch.p[i].t[0]),
                                         _mm_mul_ps(plane.Nv.t[1],
                                                    patch.p[i].t[1])),
                              _mm_mul_ps(plane.Nv.t[2],patch.p[i].t[2]));
   const sse_t d_f = _mm_sub_ps(d,plane.dv);
   Nv_dist[i] = d_f;
   signsV |= _mm_movemask_ps(d_f) << (4*i);
 }
 if (signsV == 0xffff || signsV == 0) return false;
 return true;
}
```

*Figure 6.15: Initialization of the sixteen 2D control points of patch $P'(u, v)$. The control points are the distances $d_{ij}$ of the original control points to the two ray planes (stored in the SSE-suitable SOA format). If the new control points lie completely in one half space with respect to a ray plane, the new patch cannot contain the origin and the function returns false (early exit).*

### Origin Test

The test whether the current patch $P'(u, v)$ includes the origin, or not, is basically the same as the previously mentioned sign test of all distances. Since this test is performed for every Bézier clipping iteration it needs to be as fast as possible. Therefore, a simplified version is chosen. This test first computes the extremal distances (minimum and maximum) with respect to

```
inline bool ContainsOrigin()
{
  sse_t minU = Nu_dist[0];
  sse_t maxU = Nu_dist[0];
  for (unsigned int i=1;i<4;i++)
    {
      minU = _mm_min_ps(minU,Nu_dist[i]);
      maxU = _mm_max_ps(maxU,Nu_dist[i]);
    }
  if ((_mm_movemask_ps(maxU) == 0xf) ||
      (_mm_movemask_ps(minU) == 0x0)) return false;

  sse_t minV = Nv_dist[0];
  sse_t maxV = Nv_dist[0];
  for (unsigned int i=1;i<4;i++)
    {
      minV = _mm_min_ps(minV,Nv_dist[i]);
      maxV = _mm_max_ps(maxV,Nv_dist[i]);
    }
  if ((_mm_movemask_ps(maxV) == 0xf) ||
      (_mm_movemask_ps(minV) == 0x0)) return false;
  return true;
}
```

*Figure 6.16: Checking if the 2D patch $P'(u, v)$ contains the origin by applying the plane test only to extremal values of the control point matrix. For every column of the control point matrix, the minimum and maximum is computed in parallel and the corresponding signs are compared. If all maximum, respectively minimum, values have negative respectively positive signs, the patch cannot contain the origin.*

the columns of the control point matrix. The origin test is performed in a second step by simply testing those extremal points, as shown in Figure 6.16.

### Computing $\mathbf{L}_u$ and $\mathbf{L}_v$

Reducing the parametric domain in either the $u$ or $v$ direction makes it necessary to compute the two lines $L_u = \frac{1}{2}(P_{30} - P_{00} + P_{33} - P_{03})$ and $L_v = \frac{1}{2}(P_{03} - P_{00} + P_{33} - P_{30})$. $L_u$ ($L_v$) should be roughly perpendicular to direction $u$ ($v$). Given the nature of Bézier clipping, it is possible that either $L_u$ or $L_v$ become zero. In this case, the zero vector will be replaced by a vector which is perpendicular to the non-zero vector. As computing $L_u$ and $L_v$ is rather simple, code is omitted.

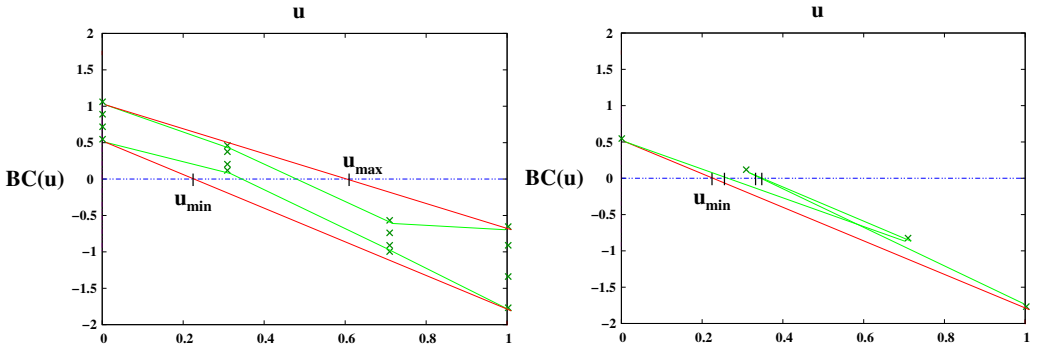



*Figure 6.17: Left Image: When computing the intersection of the convex hull with the u-axis, only the extremal intersections $u_{min}$ and $u_{max}$ need to be considered. Right Image: In the case of multiple intersections, only the absolute maximum, respectively minimum, of the intersections with the u-axis is determined.*

### Computing the Convex Hull

Assuming parametric direction $u$ has been chosen, the new control point matrix $d_{ij}^{'}$ is set by computing the distances $d_{ij}$ to $L_u$. Because of the equidistant partition in $u$, all $d_{ij}^{'}, 0 \leq i \leq 3$ share the same $u_i = i * \frac{1}{3}$ value. Therefore, the minimum $dmin_i^{'}$ and maximum $dmax_i^{'}$ of $d_{ij}^{'}, 0 \leq i \leq 3$ are determined for every $v_i$. Computing $v_i$ can be implemented efficiently using SSE instructions, which allows all four $dmin_i^{'}$ and $dmax_i^{'}$ to be stored within one register. An early exit can be performed, simply by testing the four $dmin_i^{'}$ in terms of matching positive signs and $dmin_i^{'}$ in terms of matching negative signs.

Starting from $dmin_i^{'}$ respectively $dmax_i^{'}$, the convex hull with respect to a possible intersection at the $u$-axis is constructed. Note that an intersection with the $u$-axis can only exist if the signs of $dmin_i^{'}$ ($dmax_i^{'}$) do not match. For all possible intersections with the $u$-axis, only the extremal intersections $u_{min}$ and $u_{max}$ need to be considered, as shown in the left image of Figure 6.17.

For the four points $dmin_i^{'}$ ($dmax_i^{'}$), there exists only a maximum of four intersections with the $u$-axis, as shown in right image of Figure 6.17. Note that not all have a valid intersection with the $u$-axis. Based on the signs of the four points $dmin_i^{'}$ ($dmax_i^{'}$), it is possible to precompute the indices $i_{start}$ and $i_{end}$ referring to the $y$ coordinate and the $u$ values for the start and end points of all four line equations.

Taking the signs of either $dmin_i^{'}$ or $dmax_i^{'}$, one can easily assemble all four possible line equations in the SOA format by performing table look-ups. The potential intersections can now be efficiently computed using SSE in-

```
struct IntersectionTableEntry // sixteen precomputed entries
{
  unsigned int sy[4];
  unsigned int ey[4];
  float dx[4];
  float sx[4];
};
// --------------------------------
sse_t dist[4];
const sse_t nx = _mm_set_ps1(du.y);  // normal to Lu
const sse_t ny = _mm_set_ps1(-du.x);

for (unsigned int i=0;i<4;i++)
  dist[i]  = _mm_add_ps(_mm_mul_ps(nx,Nu_dist[i]),
                        _mm_mul_ps(ny,Nv_dist[i]));

sse_t min_dist = _mm_min_ps(_mm_min_ps(dist[0],dist[1]),
                            _mm_min_ps(dist[2],dist[3]));
sse_t max_dist = _mm_min_ps(_mm_max_ps(dist[0],dist[1]),
                            _mm_max_ps(dist[2],dist[3]));
if (_mm_movemask_ps(min_dist) == 0x0 ||
    _mm_movemask_ps(max_dist) == 0xf) return false;

if ((minMask & 0x9) != 0x9) {
 const unsigned int index = ((mask & 0x8) ? ~mask : mask) & 0x7;
 const IntersectionTableEntry &entry = convexHullIntersectionTable[index];
 const sse_t sy = _mm_setr_ps(ptr[(int)entry.sy[0]],ptr[(int)entry.sy[1]],
                              ptr[(int)entry.sy[2]],ptr[(int)entry.sy[3]]);
 const sse_t ey = _mm_setr_ps(ptr[(int)entry.ey[0]],ptr[(int)entry.ey[1]],
                              ptr[(int)entry.ey[2]],ptr[(int)entry.ey[3]]);
 const sse_t denom = Inverse(_mm_sub_ps(ey,sy));
 const sse_t s = _mm_add_ps(_mm_mul_ps(_mm_mul_ps(*(sse_t*)entry.dx,sy),
                                       denom),*(sse_t*)entry.sx);
 const float s_min = _mm_cvtss_f32(sseHorizontalMin(s));
 const float s_max = _mm_cvtss_f32(sseHorizontalMax(s));
 minS = MIN(minS,s_min);
 maxS = MAX(maxS,s_max);
}
if (((~maxMask) & 0x9) != 0x9) {
// ...
}
```

*Figure 6.18: Given a maximum of four intersections, the indices for the four line representations are precomputed, allowing the intersections to be efficiently computed in parallel. Horizontal operations then return the minimum and maximum of the 'u' intersections.*

structions (see Figure 6.18). Taking the minimum (maximum) of the four $u$ intersections, one can obtain $u_{min}$ ($u_{max}$). Note that the intersection computation needs to be applied only for a subset of the sixteen possible sign combinations.

```
inline void deCasteljauVertical(BicubicBezierPatch2D &dest,
                                const sse_t alpha,
                                const sse_t beta) const
{
 dest.Nu_dist[0] = Nu_dist[0];
 const sse_t p10x = _mm_add_ps(_mm_mul_ps(alpha,Nu_dist[0]),
                               _mm_mul_ps(beta,Nu_dist[1]));
 dest.Nu_dist[1] = p10x;
 const sse_t p11x  = _mm_add_ps(_mm_mul_ps(alpha,Nu_dist[1]),
                                _mm_mul_ps(beta,Nu_dist[2]));
 const sse_t p20x  = _mm_add_ps(_mm_mul_ps(alpha,p10x),
                                _mm_mul_ps(beta,p11x));
 dest.Nu_dist[2] = p20x;
 const sse_t p12x  = _mm_add_ps(_mm_mul_ps(alpha,Nu_dist[2]),
                                _mm_mul_ps(beta,Nu_dist[3]));
 const sse_t p21x  = _mm_add_ps(_mm_mul_ps(alpha,p11x),
                                _mm_mul_ps(beta,p12x));
 const sse_t p30x  = _mm_add_ps(_mm_mul_ps(alpha,p20x),
                                _mm_mul_ps(beta,p21x));
 dest.Nu_dist[3] = p30x;

 // similar for Nv_dist[i]
 // ...
}
```

*Figure 6.19: The 2D de Casteljau algorithm for subdividing in the vertical direction. 'alpha' and 'beta' are the linear factors used for computing the linear combinations.*

### Subdivision

Starting from a reduced parametric interval $[u_{min}, u_{max}]$, the corresponding control point matrix needs to be computed. Therefore, two de Casteljau subdivisions are executed using the code shown in Figure 6.19.

Note that the linear factors used for the linear combinations within the de Casteljau algorithm need to be adjusted: $u_{min}$,$1 - u_{min}$ for the first subdivision, and $\frac{u_{max}-u_{min}}{1.0f-u_{min}}$ for the second subdivision.

| Step | CPU Cycles |
|---|---|
| Initialization 3D to 2D | 74 |
| Origin Test | 70 |
| Computing $L_u$ and $L_v$ | 50 |
| Computing Convex Hull | 218 |
| Subdivision | 100 (200) |

*Table 6.4: Average number of CPU cycles for each of the core operations for Bézier clipping. The two most expensive operations are the determination of the convex hull including intersection computation, and the related subdivision. Note that subdivision in 'u' direction requires additional shuffle instructions, thereby roughly doubling the costs.*

### 6.6.3 Discussion

Table 6.4 shows the average number of cycles for different core operations. Almost all core operations require less than 100 cycles. The cost, in particular, for computing the convex hull and for patch subdivision, could be further reduced as soon as processor architectures offer better support for horizontal and shuffle operations.

The Bézier clipping algorithm in its original form does not depend on an additional per-patch spatial index structure as required by the Newton iteration. Similar to the uniform refinement approach, it only requires the original control point matrix, which makes it very memory efficient. Nevertheless, the Bézier clipping algorithm could also benefit from a patch acceleration structure to speed up convergence. Restricting the parametric starting domain associated to an index structure entry would immediately reduce the number of required iterations.

Another difficulty is that the Bézier clipping algorithm is rather sensitive to the relation of the two lines $L_u$ and $L_v$. Ideally, these two vectors should be perpendicular to each other. In practice, however, the angle between them can be rather small. In this case, only a low reduction of the parametric domain can be achieved, resulting in an increased number of iterations. It would be interesting to implement the approach proposed by Efremov et al. [Efremov05] for computing more suitable $L_u$ and $L_v$ vectors.

## 6.7 Summary of Intersection Algorithms

Except for the standard Newton iteration-based intersection, none of the described approaches depend on an additional per-patch spatial index structure.

However, Bézier clipping or Newton iteration using the Krawczyk operator might benefit from a structure of this kind. Such an index structure would restrict the parametric starting region, thereby improving convergence of the intersection algorithm. Moreover, if an intersection algorithm requires many iterations before pruning a patch, a per-patch index structure can help to perform the pruning test earlier (in the traversal phase). On the other hand, the increased number of traversal steps and memory accesses can offset the benefit gained from a higher convergence speed.

Profiling tools such as VTune [Intel04] show that for all approaches which do not rely on additional spatial index structures, the intersection sequence is compute-bound. A stack of subdivided patches (or parametric intervals), as required by most approaches, comprises only a few KB. Therefore, stack data completely resides in the cache hierarchy.

The only point where cache misses appear is the point of accessing patch data. However, most of those (L2) cache misses can be avoided by prefetching data for the subsequent patches while intersecting the current one. Since the memory latency is usually smaller than the costs of patch intersection, memory prefetching works very well in this situation.

Comparing the implementation complexity of the different intersection algorithms shows that the intersection based on uniform refinement employ the shortest core routines and the simplest control flow. A slightly more complex algorithm is the Newton iteration with an additional spatial index structures over sub-patches, while Bézier clipping is the most complex intersection algorithm. Newton iteration using the Krawczyk operator is as complex as Bézier clipping. Whenever short construction times are mandatory, e.g. for supporting dynamic scenes (see Chapter 7), one should choose an intersection algorithm that does not depend on additional per-patch spatial index structures. Furthermore, if limited accuracy is acceptable, uniform refinement typically provides the highest performance (see Section 6.10). For actual performance statistics for ray tracing complete bicubic Bézier scenes, see Section 6.10.

When it comes to supporting ray bundles, uniform refinement and Newton iteration with a sub-patch spatial index structure provide a straightforward extension. Integrating ray bundles into Bézier clipping is more difficult because the subdivision has to be done on a per ray basis. Moreover, Bézier clipping requires numerous temporary data updates, which is difficult to realize for many rays in parallel. Therefore, it is likely that the benefit of computing the intersection for multiple rays in parallel will be offset by the additional overhead.

For a Newton iteration-based intersection algorithm using the Krawczyk operator the support of ray bundles is non-trivial. As each ray uses its own

parametric interval, the patch clipping step must be performed individually for each ray.

On the other hand, it might be beneficial to split the intersection algorithm into a serial and a parallel part: This means evaluating the Krawczyk operator for each ray in a serial process, but performing the computation for the subsequent Newton iterations in parallel.

## 6.8 Spatial Index Structures for Patches

Having a fast primitive intersection code is only a first step, because typical scenes are comprised of many primitives which will require the construction of a spatial index structure to reduce the number of ray-patch intersection tests per ray. Compared to a simple ray-triangle intersection, the cost for a ray-patch intersection is significantly higher. This means that a small number of ray-patch intersection tests per ray is essential for achieving high performance, which puts certain quality requirements on the spatial index structure used. In the past, researchers mostly used bounding volume hierarchies [Rubin80, Kay86, Haines91, Smits98] as the spatial acceleration structure for reducing the number of ray-patch intersection tests. Because of the very fast traversal code (see Chapter 4) and its adaptation ability with respect to geometric distribution, a kd-tree is chosen as the spatial index structure.

A kd-tree over patches is built based on the axis-aligned bounding boxes (AABB) for each patch, ignoring the patch shape itself. Even though this simplification is less accurate, it is both simple and fast. To enhance the quality of the kd-tree, a surface area heuristic (SAH) [Havran01, Wald04, Benthin04] is applied to carefully position the splitting plane. This results in better quality (fewer intersection tests per ray), compared to a standard kd-tree construction algorithm. For a more detailed discussion of how to construct kd-trees out of axis-aligned bounding boxes, see Section 7.1.

## 6.9 Trimming Curves

Trimming curves are a common method for overcoming the topologically rectangular limitations of patches. Trimming curves, which are defined in two-dimensional parameter space, are mostly used when designers wish to cut out sections from patches.

Similar to the conversion of NURBS to bicubic Bézier patches (see Section 6.2), an arbitrary source representation for a trimming curve is converted

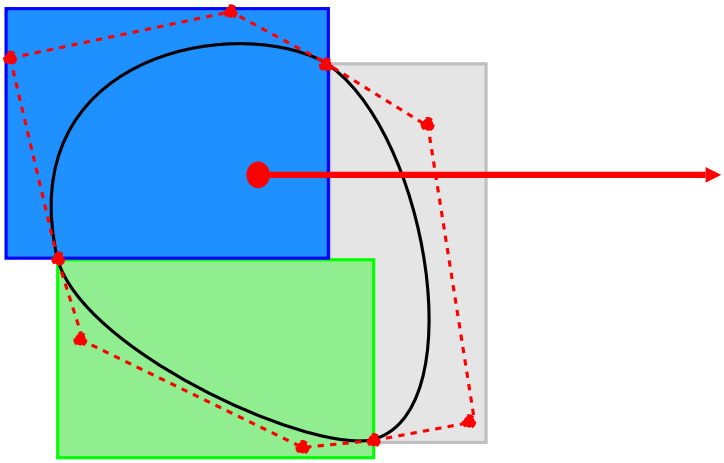


*Figure 6.20: Example trimming curve consisting of three bicubic Bézier segments. For each segment the axis-aligned bounding box (marked blue, green, and gray) is used as the construction primitive for the kd-tree. The kd-tree as the spatial index structure allows for reducing the number of potential segments which are passed to the ray-segment intersection test. In order to determine whether a given point is inside (outside) a trimming curve, the number of intersections between a virtual ray (marked red) and the trimming curve segments is counted. In the case of an odd (even) number, the corresponding point is inside (outside) with respect to the curve.*

to a sequence of bicubic Bézier curve segments. Restricting the degree of the curve to three allows for a fast and less complex implementation.

Trimming curves have two important properties: they are closed and they must not overlap. Exploiting these properties allows for constructing a trimming curve hierarchy. Additionally, each trimming curve saves its orientation, which defines whether the inside or outside (with respect to the curve) has to be removed.

In the case one or more trimming curves are attached to a given patch, and the corresponding ray-patch intersection returns a hit, the parametric hit coordinates are passed to a trimming curve (hierarchy) test. If a trimming curve hierarchy exists for the given patch, the hierarchy is traversed as described in [Martin00]. An intersection point lies inside (outside) with respect to a given trimming curve, if the number of intersections, between a ray starting from the intersection point towards a point outside the trimming curve, is odd (even). The task is therefore to count the intersections between a "virtual ray" and the underlying bicubic Bézier segments. In order

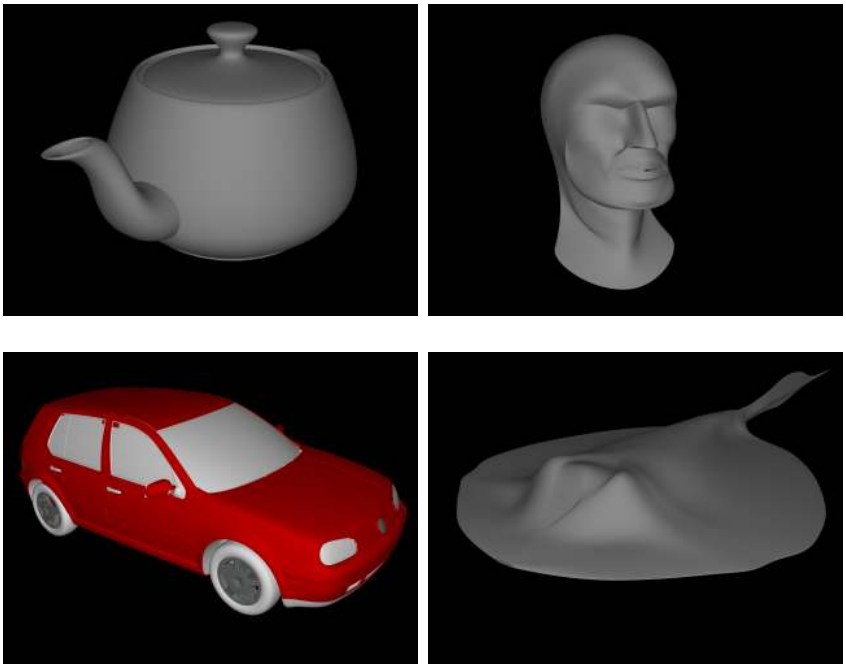


*Figure 6.21: A set of test scenes that have been converted into bicubic Bézier patches. The number of Bézier patches is* 32 *for the "Teapot",* 915 *for the "Head",* 20,257 *for the "VW Golf" and* 1,160 *for the "Stingray". The "VW Golf" scene contains* 8150 *trimming curves, corresponding to* 119,580 *bicubic Bézier curve segments.*

to speed up intersection calculation, a two-dimensional kd-tree is constructed over all segments. This allows for reducing the number of potential segments that need to be tested for intersection. For the actual kd-tree construction, a similar approach as for patches is chosen: The axis-aligned bounding box (now in 2D) of each curve segment is used as construction primitive (see Figure 6.20). In order to compensate for the coarser hull approximation, an SAH is applied to improve kd-tree quality. For the real ray-curve intersection test, any of the algorithms shown in Section 6.2 is suitable. Typically, uniform refinement with a fixed number of refinement steps offers sufficient speed and accuracy. As all operations are now performed in 2D, roughly 30% of the respective core routine costs are saved as compared to the operation costs in 3D.

## 6.10 Results

This section will provide performance statistics and analyses for ray tracing bicubic Bézier scenes. For testing purposes, four scenes with different complexity have been used (see Figure 6.21). Note that only the "VW Golf" scene contains trimming curves.

Comparing the different intersection approaches is difficult. This is especially true when it comes to comparing uniform refinement with approaches based on analytical solutions. Therefore, the algorithmic analysis is split into two parts: The first part covers intersection for uniform refinement while the second part deals with the analytical algorithms.

Regardless of the intersection algorithm, ray tracing of Bézier patches requires less time in kd-tree traversal than for triangles. Even if an additional index structures over sub-patches is used, typically only 20-30% of the time for tracing a single ray is spent in kd-tree traversal, while ray-patch intersection takes up 70-80% of the time. The performance impact of fast traversal code is therefore less than for ray tracing triangles (typically traversal 70%, intersection 30%). The simple reason for this is that total performance is exclusively determined by the intersection step. Even though the cost for intersection already starts to dominate even in the triangle case, for Bézier patches the cost discrepancy between traversal and intersection is even more significant. In the case no sub-patch index structures are used, a kd-tree over patches is shallower as compared to the triangle case. This results in 40-50% fewer traversal steps in total, additionally shifting the dominating cost factor towards intersection.

| Rays | Teapot | VW Golf | Head | Stingray |
|---|---|---|---|---|
| 1 | 378760 | 704965 | 219687 | 440892 |
| 16 | 6.71% | 10.21% | 10.6% | 8.74% |

*Table 6.5: Average number of patches loaded per frame at a resolution of* $640 \times 480$*, casting only primary rays. Compared to tracing single rays, tracing ray bundles of sixteen rays reduces patch accesses to less than 10% of the number for individual rays.*

Extending single ray traversal to ray bundle traversal allows for exploiting the same benefits as those of the triangle case. In particular, any fast traversal code can be reused directly. Only the code for triangle intersections has to be replaced by patch intersection code. Moreover, a similar bandwidth reduction as for triangles can be achieved, because neighboring rays are likely to access the same patches. The loaded patch data can therefore be reused for all rays

within the bundle. This reduces the total number of patch accesses per frame, and therefore the required bandwidth to memory. Table 6.5 shows that for $4 \times 4$ ray bundles, the bandwidth required for loading patch data can be reduced to less than 10% of the original amount. For scenes with additional sub-patch index structures, the reduction is typically smaller (depending on the depth of the sub-patch index structure).

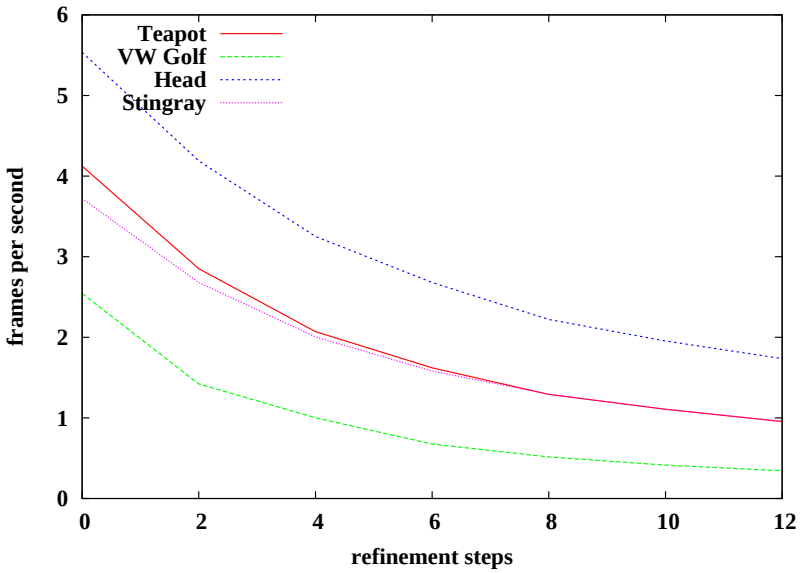



*Figure 6.22: (Single ray) performance (in fps) for Bézier scenes as a function of the number of patch refinement steps. Each refinement step corresponds to a de Casteljau subdivision step in one of the two parametric directions. All experiments were performed at a resolution of* $640 \times 480$ *pixels on a single Pentium-IV 2.2 GHz processor. Using* $4-6$ *refinement steps, an intersection based on uniform refinement is able to achieve interactive performance on a single processor.*

### 6.10.1 Uniform Refinement Results

In order to demonstrate the impact of an increasing refinement level, Figure 6.22 illustrates the performance in frames per second for all four test scenes, casting a single primary ray per pixel. All tests were run on a single Pentium-IV 2.2 GHz processor at a resolution of $640 \times 480$. Even for a high number of refinement steps, interactive performance is realized. The achieved accuracy depends to a great extent on the patch shape itself, but for most

| Rays | Teapot | VW Golf | Head | Stingray |
|---|---|---|---|---|
| 1 | 2.0 | 1.01 | 3.23 | 1.89 |
| 16 | 16.14 | 5.72 | 12.93 | 9.02 |
| Speedup | 8.07x | 5.67x | 4.0x | 4.77x |

*Table 6.6: Performance comparison between tracing single rays and tracing bundles of sixteen rays, in frames per second. All scenes are rendered at a resolution of* $640 \times 480$*, casting only primary rays and with a refinement level of* 4*. Tracing ray bundles of sixteen rays results in a speedup factor of* $4 - 8$*, compared to tracing single rays. For the "VW Golf" scene, no evaluation of trimming curves has been applied.*

applications a refinement level of $4 - 6$ already provides smooth surfaces.

To further improve performance, the single ray intersection code was extended in order to support ray bundles. Uniform refinement, in particular, can benefit enormously from ray bundles because the patch subdivision cost can be efficiently amortized over all rays within the bundle. As for the triangle case, a bundle size of 16 rays has been found to be most beneficial.

A comparison between the fastest code for tracing single rays and the fastest code for tracing ray bundles shows that a speedup factor of $4 - 8$ is achieved through bundling (see Table 6.6). By increasing coherence within the ray bundle, e.g. by rendering at resolutions higher than $640 \times 480$, this factor can be increased even further.

| Rays | Teapot | VW Golf | Head | Stingray |
|---|---|---|---|---|
| $640 \times 480$ | 8.07 | 5.67 | 4.0 | 4.77 |
| $800 \times 600$ | 8.25 | 6.61 | 4.48 | 5.17 |
| $1024 \times 1024$ | 8.6 | 8.42 | 5.64 | 6.28 |
| $1600 \times 1200$ | 8.7 | 8.5 | 5.7 | 6.41 |
| Speedup | 7.8% | 49.91% | 42.5% | 34.38% |

*Table 6.7: Speedup improvement for tracing ray bundles compared to tracing single rays with increasing resolution. The more coherent the rays are, the higher the speedup improvement (up to* 49.91%*).*

Table 6.7 shows the increase of the speedup factor for all test scenes when rendered at different resolutions. At a resolution of $1024 \times 1024$, the increased coherence within the ray bundle results in a 49.91% speedup as compared to rendering at a resolution of $640 \times 480$.

Higher coherence does not only affect traversal performance but, above all, intersection performance. The more coherent the rays are, the higher the decision coherence within uniform refinement. The largest performance gain is achieved for the "VW Golf" scene because the scene consists of many small patches which negatively impact decision coherence within traversal and intersection when rendering at low resolutions.

As a result, intersection based on uniform refinement is able to fully exploit ray bundle coherence. If high accuracy requirements are not mandatory (depending on the scene), a fast uniform refinement implementation is able to provide interactive performance even on a single processor.

### 6.10.2 Analytical Intersection Results

As analytical intersection algorithms iteratively converge to the right solution, a fixed accuracy threshold is used. More precisely, if a computed solution yields an error which lies below a predefined value, the solution is accepted as valid.

| Rays | Teapot | VW Golf | Head | Stingray |
|---|---|---|---|---|
| Patches | 32 | 20,257 | 915 | 1160 |
| Sub-patches | 2016 | 255,188 | 15,538 | 4,650 |
| IRays | 273,663 | 215,821 | 151,365 | 252,399 |
| TSteps | 23.59 | 35.11 | 35.70 | 22.88 |
| PI/IRay | 1.01 | 2.33 | 1.67 | 1.51 |
| NI/IRay | 2.59 | 8.04 | 5.61 | 4.93 |
| fps | 2.5 | 1.35 | 2.4 | 2.0 |

*Table 6.8: Single ray statistics for Newton iteration-based intersection using an additional per-patch index structure to restrict the parametric search domain. Only sufficiently small parametric domains (sub-patches) will ensure convergence of the Newton iteration. Compared to the original number of patches, the number of sub-patches is increased by a factor of* 4 *to* 60*. Each ray invokes roughly* $1-3$ *intersection steps, resulting in* $2-8$ *Newton iterations per ray. All scenes were rendered at a resolution of* $640 \times 480$ *on a Pentium-IV 2.2 GHz, with an accuracy threshold of* $1E-3$*. "IRays": number of rays that performed an intersection step,"TSteps": required traversal steps per ray,"PI/IRay": number of patch intersections per "IRay", "NI/IRay": number of Newton iterations required per "IRay".*

Table 6.8 shows statistics for a Newton iteration-based patch intersection. For this case, the kd-tree is constructed over sub-patches to restrict the

parametric search domain. The Newton iteration will only converge if the parametric starting region is sufficiently small. As convergence depends to a great extent on patch curvature, a simple criterion based on the deviation to a bilinear patch is used to terminate the generation of sub-patches. Depending on the scene, 4 to 60 times more sub-patches than patches have to be created in order to ensure convergence. Note that for sub-patches, no actual patch data needs to be stored, but only a reference to the corresponding parametric domain as well as a reference to the original patch data. All required information can be efficiently stored in a 32-byte structure [Geimer05], requiring 7.78 MB of additional storage space for 215, 821 sub-patches of the "VW Golf" scene. As the kd-tree is constructed over sub-patches, the corresponding storage space increases from 3.14 MB for 20, 257 patches to 62.36 MB for 255, 188 sub-patches.

As illustrated in Table 6.8, one ray requires an average of $22-35$ traversal steps and $1-3$ intersection tests. Each intersection step requires at least one Newton iteration, resulting in $2-8$ Newton iterations per ray on average. Given the small number of required iterations and the impact of the efficient SSE implementation (see Section 6.4), interactive performance even on a single processor is achieved.

| Rays | Teapot | VW Golf | Head | Stingray |
|---|---|---|---|---|
| Patches | 32 | 20257 | 915 | 1160 |
| IRays | 273,663 | 215,821 | 151,365 | 252,399 |
| TSteps | 10.22 | 24.64 | 23.12 | 16.63 |
| PI/IRay | 1.31 | 3.26 | 2.08 | 1.71 |
| BC/IRay | 9.96 | 2.03 | 2.31 | 2.48 |
| fps | 0.8 | 1.25 | 2.37 | 1.73 |

*Table 6.9: Single ray statistics for Bézier clipping-based intersection. Each ray invokes* $1-3$ *intersection steps, resulting in* $2-10$ *Bézier clipping steps per ray. All scenes were rendered at a resolution of* $640 \times 480$ *on a 2.2 GHz Pentium-IV, with a solution accuracy of* $1E-3$*. "IRays": number of rays that performed an intersection step, "PI/IRay": number of patch intersections per "IRay", "BC/IRay": number of Bézier clipping iterations required per "IRay".*

Intersection based on Bézier clipping, as illustrated in Table 6.9, does not depend on additional data structures in order to ensure convergence. As a result, the kd-tree is significant smaller, resulting in up to 50% fewer traversal steps compared to the sub-patch-based intersection. For most scenes,

only $2-3$ Bézier clipping iterations are required. As with Newton iteration-based intersection, the "Teapot" with its highly curved patches requires significantly more iterations (more pre-subdivisions) than all other scenes. A performance comparison between the two intersection approaches shows that the costs for Bézier clipping cannot be compensated for by the smaller number of traversal steps. Bézier clipping provides a slightly lower performance than Newton iteration-based intersection but, on the other hand, requires no additional spatial index structures.

| Rays | Teapot | VW Golf | Head | Stingray |
|---|---|---|---|---|
| Patches | 32 | 20257 | 915 | 1160 |
| IRays | 273,663 | 215,821 | 151,365 | 252,399 |
| TSteps | 10.22 | 24.64 | 23.12 | 16.63 |
| PI/IRay | 1.31 | 3.26 | 2.08 | 1.71 |
| KI-Clip/IRay | 12.76 | 6.01 | 4.53 | 4.65 |
| KI-Interval/IRay | 44.23 | 22.82 | 14.49 | 14.89 |
| NI/IRay | 0.53 | 1.76 | 0.87 | 0.83 |
| fps | 0.21 | 0.25 | 0.52 | 0.43 |

*Table 6.10: Single ray statistics for Newton iteration-based intersection using the Krawczyk operator. All scenes were rendered at a resolution of* $640 \times 480$ *on a 2.2 GHz Pentium-IV, with a solution accuracy of* $1E-3$. *"IRays": number of rays that performed an intersection step, "PI/IRay": number of patch intersections per "IRay", "KI-Clip/IRay": number of Krawczyk operator evaluations per "IRay", using a clipping operation to compute the interval extension of the partial patch derivatives,"KI-Interval/IRay": number of Krawczyk operator evaluations per "IRay", using pure interval arithmetic to compute the interval extension of the partial patch derivatives, "NI/IRay": number of Newton iterations required per "IRay".*

Table 6.10 shows statistics for Newton iteration-based intersection using the Krawczyk operator. The algorithm does not rely on additional per-patch spatial index structures, which implies the same number of traversal and intersection steps per ray as for Bézier clipping. In terms of iterations for the Krawczyk-Moore test, the higher accuracy of the interval extension of the partial derivatives makes for a reduction of up to a factor of four as compared to using pure interval arithmetic.

As the "Teapot" scene contains highly curved patches, the Krawczyk operator using clipping operations still needs over 12 iterations per ray, as compared to $4-6$ for the other scenes, to provide safe starting intervals.

Moreover, the safe starting intervals are typically very small which allows for fast convergence of the Newton iteration. For the four test scenes, only $0.5 - 2$ iterations per ray are required in order to compute the final intersection. However, total performance is lower than with the standard Newton iteration-based algorithm. The reasons behind the lower performance are the costly operations required for evaluating the Krawczyk operator. Further optimization should thus concentrate on reducing the cost for these operations.

As only the "VW Golf" scene includes trimming curves, the impact of trimming curve evaluation could only be measured for this scene. Depending on the view, additional trimming curve tests cause a total performance decrease of $10 - 30\%$ (compared to the performance statistics of Tables 6.8, 6.9, and 6.10).

Extending ray-patch intersection algorithms to efficiently support ray bundles is difficult. Certain algorithms, e.g. Bézier clipping or Newton iteration using the Krawczyk operator, cannot be easily extended to efficiently support ray bundles because these algorithms require many complex operations as well as a complex control flow. Implementing these operations for ray bundles is likely to introduce additional implementation overhead. This is not the case for the standard Newton iteration-based intersection. Table 6.11 illustrates statistics for $2 \times 2$ ray bundles using standard Newton iteration-based intersection. At a resolution of $1024 \times 1024$, ray bundles allow for achieving a speedup factor of 1.58 to 2.13, compared to tracing single rays. An interesting point is that the number of iterations for ray bundles is only slightly increased, even though the exit point is not taken unless all four rays fulfill the exit criteria. This indicates high decision coherence within the algorithm.

The speedup for ray bundles could be even higher, but the small number of registers and the limited floating point capability of current CPU architectures have a negative impact on intersection performance. These limitations significantly affect bundles larger than $2 \times 2$ rays. For such larger bundles, standard Newton iteration-based intersection is sub-optimal. For every accessed sub-patch, a complete intersection test for all rays within the bundles, has to be performed (as long as not all bundles have a valid intersection). Many small sub-patches (caused by patch curvature), increase the probability that a high number of sub-patches, will need to be accessed, until all rays within the bundle have found a valid intersection.

Assuming future CPU architectures will have a higher number of registers and a improved support for floating point computations, a Newton iteration-based intersection algorithm using the Krawczyk operator might then be beneficial. The Krawczyk operator would return, for each ray within the bundle, either a valid starting interval or the statement that the ray will

| Rays | Teapot | VW Golf | Head | Stingray |
|---|---|---|---|---|
| TSteps | 26.71 | 35.33 | 35.67 | 20.83 |
| PI/IRay | 1.25 | 2.47 | 1.71 | 1.66 |
| NI/IRay | 3.28 | 8.31 | 5.29 | 5.19 |
| fps | 0.68 | 0.41 | 0.56 | 0.54 |
| TSteps $2 \times 2$ | 26.62 | 35.41 | 32.69 | 20.9 |
| PI/IRay $2 \times 2$ | 1.35 | 3.11 | 2.04 | 1.91 |
| NI/IRay $2 \times 2$ | 3.76 | 11.5 | 5.29 | 6.40 |
| fps | 1.45 | 0.65 | 0.93 | 0.98 |
| Speedup | 2.13x | 1.58x | 1.66x | 1.81x |

*Table 6.11: Comparison of Newton iteration-based intersection for single rays against bundles of* $2 \times 2$ *rays. All scenes were rendered at a resolution of* $1024 \times 1024$*, with a solution accuracy of* $1E-3$*. Even though the exit point within the Newton iteration is not taken unless the criteria are fulfilled by all four rays, the required number of Newton iterations is only slightly increased. Therefore, a performance speedup factor of* 1.58 *to* 2.13 *is achieved, compared to casting single rays.*

miss the patch. Because of the complex control flow, it might prove useful to perform these operations sequentially. The result of this operation sequence would be an array of starting intervals as well as a bit-mask to indicate whether a ray will intersect a given patch or not. Based on these values a parallel Newton iteration could be performed for all rays within the bundle. The iteration would stop once all *valid* rays (for which a safe starting region has been identified) indicate a valid intersection.

Obviously, all traversal algorithms from Section 4.2, e.g. reducing traversal steps by finding better kd-tree entry points, could be applied here. However, most of the total time is spent in intersection, which limits the positive impact of a smaller number of traversal steps. The main goal should therefore be to further optimize intersection.

## 6.11 Application

Computer-Aided Design and Virtual Reality are becoming increasingly important tools for industrial design applications. Large and high-end engineering projects, in particular, such as cars or airplanes, are already being designed on an almost entirely digital basis, as the cost for building physical mockups of such objects is very high.

*Figure 6.23: Top left: Mercedes C-class model directly exported from a CAD system and converted into* 319,340 *bicubic Bézier patches and* 1.46 *million (bicubic Bézier) trimming curves. The patches are ray traced directly without triangulation. Top right: In order to illustrate the complexity of the Mercedes model, a random color is assigned to each patch. Center left: The model with advanced shading for glass and car paint, combined with surrounding geometry and an HDR environment map. Center right, Bottom left, Bottom right: Close-ups demonstrating the high geometric accuracy necessary for high-quality visualization of smooth surfaces.*

CAD engineers work on the raw geometric data of the model, usually using freeform data such as NURBS surfaces [Piegl97]. In order to satisfy the requirement of interactivity, practically all of today's virtual reality (VR) systems are built on triangle rasterization hardware. With this, however, rendering complete models with high geometric accuracy is not possible, as freeform surfaces usually cannot be rendered directly, and the number of triangles generated by a high-quality tessellation is in the order of tens of millions of triangles.

The process of tessellating freeform models, which consists of converting them first to triangles and subsequently carrying out a simplification step, is currently state-of-the-art in the industry. However, this approach has two important drawbacks: First, only limited geometric accuracy is achieved and, second, preparing these special VR models takes time, so changes of the original model may take up to several days before they can be shown in a VR presentation. This often leads to decision makers looking at outdated model variants.

Thanks to the seamless integration of fast intersection implementations for freeform surfaces (see Chapter 6) into the ray tracing framework, it is now possible to deliver the level of interactive performance which is required for VR systems. Moreover, ray tracing-based rendering algorithms allow for achieving higher shading quality than rasterization-based VR systems could provide.

Figure 6.23 shows examples of a ray tracing-based VR system which utilized the fast freeform ray tracing code from Chapter 6. The model consisted of 69,067 NURBS surfaces and 392,491 2D NURBS curves forming 73,749 trimming curves, directly exported from a CAD system. The 69,067 NURBS surfaces were converted into 308,095 Bézier patches of arbitrary degree. In order to fully exploit the fast patch intersection code of the previous section, a patch degree reduction was applied, yielding 319,340 bicubic Bézier patches. Degree reduction was applied in a similar way for the 2D NURBS curves, resulting in 1.46 million bicubic Bézier curves, i.e. 5 trimming curves per patch. Besides the geometry itself, extremely complex shading, e.g. sophisticated car paint or glass shading, was applied to provide as much realism as possible.

A cluster of 15 dual-processor PCs (a mix of 3.0 GHz Intel Pentium-IVs and 2.4 GHz Opterons) was used to provide enough compute power for performing all ray tracing and shading tasks at interactive rates (see Chapter 8). This setup allowed for achieving high-quality rendering with a frame rate of up to 10 frames per second, at a resolution of $640 \times 480$. As the underlying shading system only supports single rays, neither ray bundle traversal nor ray bundle intersection were used.

Apart from the direct support of CAD data, the ray tracing-based VR prototype system shows the advantage of a decoupled ray tracing and shading process. The complete shading framework, in particular, is fully independent of the underlying geometry and was developed using triangular models. In order to render bicubic Bézier patches, only the underlying core ray tracing algorithm had to be extended to support patches.

## 6.12 Conclusions and Future Work

This chapter has shown that by optimizing algorithms and adapting them to the underlying processor architecture, interactive ray tracing of freeform surfaces can be realized even on a single processor. However, for a highly accurate analytical solution, the computing power of a single processor is not enough in order to provide high frame rates. If future processor architectures provide improved support for SSE execution, e.g. more registers and execution pipelines, all proposed intersection algorithms will benefit directly.

Even though Newton iteration-based intersection using the Krawczyk operator is the slowest algorithm, it has the important advantage of providing reliable results. Therefore, future implementations need to focus on speeding up this algorithm. It should be particularly interesting to find other more efficient ways to compute the interval extension of the partial derivatives, which is currently the limiting factor. If the patch clipping operation can be avoided, the algorithm will allow efficient implementation for ray bundles.

# Chapter 7
# Dynamic Scenes

Apart from the ray tracing computation itself, the time required for preprocessing scene data is often ignored. The most time-consuming part of preprocessing scene data is usually taken up by the construction of spatial index structure(s). For traditional high-quality off-line ray tracing, the spatial index is built from scratch for every frame. Construction costs in particular are usually amortized over the number of rays shot. For high-quality rendering, the shading operations and the number of rays shot are high. As a result, the cost for construction is negligible. The off-line approach allows for supporting arbitrary dynamic scenes because reconstruction from scratch permits arbitrary manipulation of the scene's geometry between subsequent frames. Obviously, this approach is incompatible with a realtime ray tracing system.

For a realtime ray tracing system, the assumption of negligible cost for constructing spatial index structures is no longer true. Since realtime ray tracing requires highly optimized spatial index structures, the construction phase can take up to several minutes. This makes it impossible to reconstruct the spatial index structure at interactive frame rates.

In order to still be able to support dynamic scenes, researchers [Lext01, Wald03b, Wald04] have restricted the manipulation to applying hierarchical affine transformation to rigid bodies. This approach is suitable because for most applications large parts of the scene remain static, while dynamic parts are only manipulated by affine transformation. Therefore, the geometric primitives which undergo dynamic manipulation are classified according to their transformation behavior. Primitives that share the same transformation are combined into geometric objects and for each of these objects a separate spatial index structure is built.

When the transformation of an object changes, the corresponding spatial

index structure does not need to be rebuilt. Instead, during traversal the rays are transformed to the local coordinate system of the object. As the transformation is represented as a matrix, it can be efficiently realized by a multiplication with the inverse matrix.

Wald [Wald04, Wald03b] also pointed out that the approach directly supports multiple instantiation of the same geometric object (without actually replicating the geometry). Due to the fact that an instance only requires a reference to the object and the corresponding transformation, it will need no more than a few bytes of memory. If a scene can be efficiently represented by instances of a set of objects, the required storage space can be several magnitudes lower than without instancing [Dietrich05].

Nevertheless, hierarchical transformation only works for static objects. If the geometry of the primitives changes, this approach no longer works. This category of dynamic manipulation is called *unstructured motion*, because the underlying transformation does not follow any structure. Unstructured motion is very common in the context of games, e.g. for realizing explosions, surface deformations, and character skinning. The brute force way for supporting unstructured motion is to rebuild the spatial index structure for every frame from scratch.

As argued in Section 4.1, using a high-quality kd-tree as the spatial index structure has been identified as an essential component for achieving realtime ray tracing. Therefore, the goal is to build a kd-tree as fast as possible but without significantly reducing its quality. Moreover, the kd-tree build algorithm should not be restricted to a certain primitive type. Instead, it should be capable of handling arbitrary types.

In the following, an algorithm is proposed that fulfills these requirements.

## 7.1 Rapid Construction of kd-Trees

In order to handle arbitrary primitives, a straightforward approach is chosen. Instead of handling primitives by their geometric shape, only the primitive's axis-aligned bounding box is considered.

Constructing a kd-tree based on axis-aligned bounding boxes (AABBs) essentially avoids the handling of special cases as, for example, in the triangle case. The obvious disadvantage is a coarser primitive level causing reduced kd-tree quality. In order to still achieve a good kd-tree quality (sufficiently low number of traversal and intersection steps per ray), a surface area heuristic (SAH) [Havran01, Wald04, Benthin04] is used during kd-tree construction. Unfortunately, applying an SAH-based cost function significantly increases building time, because for every potential split candidate

the cost function has to be evaluated during recursive construction.

Therefore, the focus lies on a fast algorithm for building kd-trees out of AABBs using a high-performance SAH evaluation. The coding guidelines from Section 3.2, which served to optimize kd-tree traversal and primitive intersection tests, can also be used in the context of fast kd-tree construction.

### 7.1.1 Algorithm

One of the main advantages of using AABBs as primitives is that the number of potential split candidates for a given AABB is limited. One AABB can produce one or two split candidates in each of the three coordinate dimensions, according to the minimum and maximum value in the respective dimension. Only one split candidate is created if the AABB is planar in the corresponding dimension (minimum value = maximum value). In the case of two splits, each split is marked as *opened* (minimum value) or *closed* (maximum value), in the case of one split, it is marked as *planar*.

Evaluating where to split using the SAH is straightforward: For each split $s_i$ the cost function $cost(s_i)$ is computed which corresponds to the estimated splitting cost at the particular position. After evaluating the cost for all splits, the smallest cost value is compared to the cost of not splitting at all (resulting in sequentially intersecting all remaining primitives). If the cost for the best split is less than the cost for non-splitting, the current voxel is splitted (the AABB spawned by all contained primitive AABBs) at the corresponding split position. If not, a kd-tree leaf is created based on the current set of primitives. Note that finding the 'best' split requires considering all three dimensions.

The cost function $cost(s)$ [Goldsmith87, MacDonald89, MacDonald90, Subramanian90b, Havran01, Wald04] itself is implemented as follows:

$$cost(s) = P_L(s) * \frac{SA_L(v,s)}{SA(v)} + P_R(s) * \frac{SA_R(v,s)}{SA(v)} \tag{7.1}$$

$SA_L$ and $SA_R$ compute the surface area of the current voxel $v$ limited on the left, respectively right, of the voxel by split $s$. The surface area of the current voxel $v$ is given by $SA(v)$. The ratio $\frac{SA_L(v,s)}{SA(v)}$ of surface areas corresponds to the probability (heuristic assumption) that a ray will intersect the left voxel (with respect to split $s$) [MacDonald89, MacDonald90, Subramanian90b, Havran01]. The difficult part is to efficiently compute $P_L$ and $P_R$, i.e. the number of primitives on the left respectively right side of split $s$. Note that constant costs for intersecting each primitive are assumed, which permits to simplify the equation to the form as shown above.

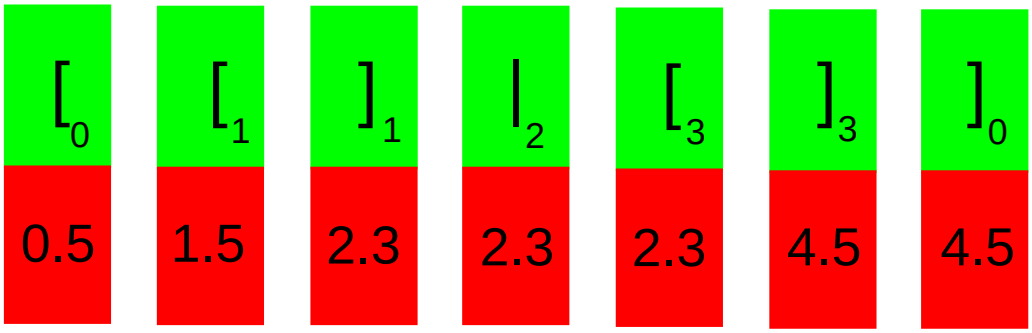



*Figure 7.1: Example of a sorted split list. The green part shows the split type (opened, planar, closed), while the red part shows the split position. In the case of equal positions, the splits are sorted in closed, planar, opened order. While sequentially iterating over all splits, the SAH can directly be evaluated on the basis of the known number of primitives to the left/right of the split. If the iteration reaches the splits with position 2.3, for example, it will first of all check for multiple candidates at the same position. If this is the case, the iteration stops at the first split which is classified as opened. Up to this point, two splits have been classified as opened, one as planar and one as closed, which corresponds to three primitives to the left of the split and two primitives to the right.*

Computing $P_L$ and $P_R$ is simplified significantly when all possible splits are sorted with respect to their position (in one determined dimension). By knowing the total number of splits and counting all closed, planar and opened splits, while sequentially evaluating the cost function for each split candidate in this ordered split list, $P_L = planar + opened$ and $P_R = splits - planar - closed$ can be easily determined (see Figure 7.1). The important point is that this evaluation only requires one sequential pass over all possible split candidates.

After determining the best split, all three split lists (which correspond to the set of AABBs) have to be inserted in the list of the left or right child node. An AABB will be inserted left (right) if the position of the corresponding closed (opened) split is less (greater) than the position of the current split candidate. All AABBs which cannot be classified via this criteria will be included in both lists. In the special case of a planar AABB that lies exactly at the position of the chosen split candidate, the AABB can either be assigned to a fixed side (with respect to the split candidate) or to the child node corresponding to the smallest voxel.

```
struct {
 float position;
 unsigned int type : 2;   // the uppermost bits for the split type
 unsigned int index : 30; // the first 30 bits for the AABB index
};
```

*Figure 7.2: The 8-byte structure efficiently represents a split candidate for kd-tree construction. The split types (opened, closed, and planar) are coded using a single 32-bit integer.*

### 7.1.2 Implementation

The split data structure consists of the split position, the split type, and the index to the corresponding AABB. As shown in Figure 7.2, all data can be efficiently represented via an 8-byte structure.

An AABB can produce two splits in each of the three coordinate dimensions, resulting in a maximum of $3 * 2 = 6$ splits (48 bytes) per AABB.

Sorting of all splits is essential for quickly computing the cost function. In order to avoid costly sorting in each kd-tree construction step, all splits are sorted once at the beginning of the kd-tree construction, and the lists of splits are kept in order during recursive kd-tree construction. The actual sorting is implemented by an optimized in-place quicksort routine, which is replaced by insertion sort once a small number of elements is reached.

The kd-tree construction algorithm consists of two steps: Creating and sorting the three split candidate lists and the recursive construction itself (see Figure 7.3).

In the case of multi-threaded kd-tree construction, the split list in each dimension is split into sub-lists according to the number of threads. Each thread sorts its sub-list independently. After each sub-list has been sorted, the sub-lists are combined hierarchically by a merge sorting step.

Besides parallel sorting, the actual kd-tree construction step can also be done in parallel. At first, a short sequence of recursive split evaluations is executed to produce an initial spatial distribution, more precisely, a set of voxels together with a split list assigned to each of them. These split lists do not depend on each other and the corresponding kd-trees can therefore be constructed in parallel. As the final step, all sub kd-trees are combined into a final kd-tree. The resulting speedup via multi-threaded construction is shown in Section 7.1.3.

A bottleneck of kd-tree construction is the required memory allocation for storing the sorted splits during recursive construction. Calling system functions (in each recursive step) for allocating memory would result in a

```
————————- Preprocessing Step ————————
empty split candidate lists spl_x, spl_y, and spl_z (index indicates dimension)
for all box b in AABB list do
   insert split candidates of box b (opened+closed or planar) into split lists spl_x, spl_y,
   and spl_z
end for
sort each split candidate list spl_x, spl_y and spl_z according to split position

——— Recursive KD-Tree Building Code: build(spl_x, spl_y,spl_z) ———-
for all split candidate list spl of spl_x, spl_y and spl_z do
   #opened = 0 // number of opened splits
   #closed = 0 // number of closed splits
   #planar = 0 // number of planar splits
   for all split candidate s of split candidate list spl do
      increase #opened, #closed, or #planar according to split type of s
      eval cost function cost(x) for s according to #opened, #closed, and #planar
      if cost(s) < cost(bestsplit) then
         bestsplit = s
      end if
   end for
end for
if cost(bestsplit) < COST_FOR_LEAF then
   for all split candidate list spl of spl_x, spl_y and spl_z do
      split spl according to bestsplit into two spl_left and spl_right
   end for
   build(spl_left-x, spl_left-y,spl_left-z) // continue recursively
   build(spl_right-x, spl_right-y, spl_right-z) // continue recursively
else
   create leaf
end if
```

*Figure 7.3: Rapid kd-tree construction algorithm based on axis-aligned bounding boxes. The algorithm basically consists of two steps: Sorting of all potential split candidates (required for fast SAH evaluation), and the recursive construction process itself. The ordered split candidate lists allow for performing the SAH evaluation of split candidates with linear complexity. As the sorting complexity is $O(n \log n)$ (quicksort) and the build complexity is $O(n \log n)$, the total complexity of the algorithm is $O(n \log n)$.*

significant performance drop. Therefore, a carefully written memory pool implementation, using a certain amount of pre-allocated memory allows for handling split lists with almost no costly system calls.

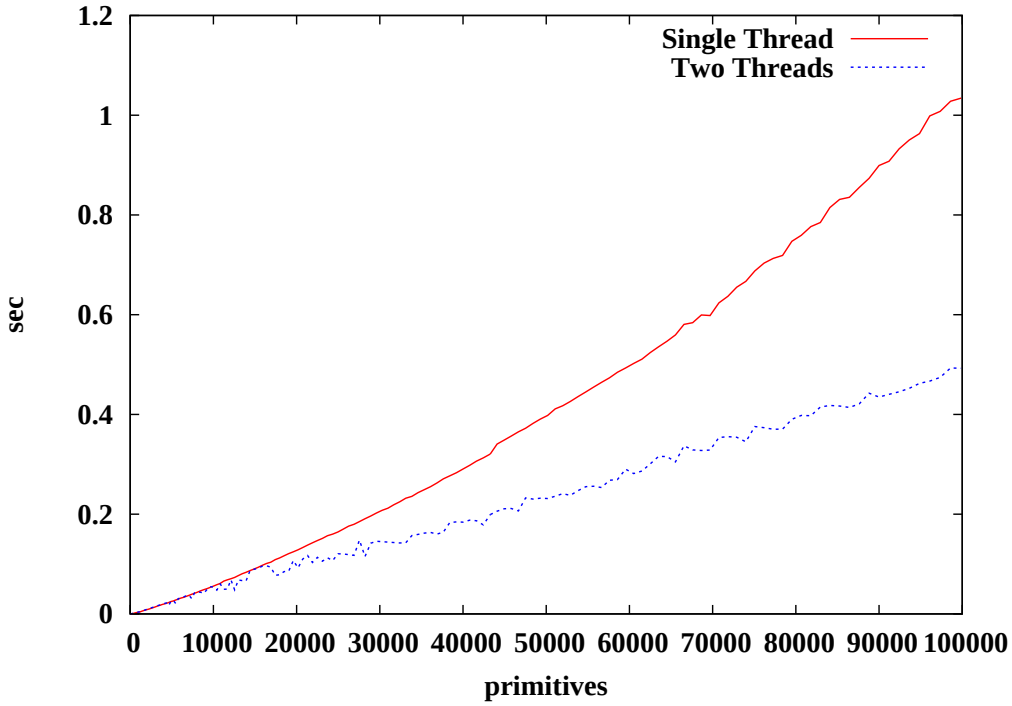



*Figure 7.4: kd-tree construction time with an increased number of primitives. Even when using costly surface area heuristics for determining the best split plane, the highly optimized implementation allows for constructing the kd-tree for* $100,000$ *primitives in less than half a second when distributing the work among two threads. Note that due to memory caching effects, the multi-threaded construction shows even a superlinear speedup.*

### 7.1.3 Results

An optimized implementation of split sorting, cost function evaluation, split list partitioning, and leaf creation makes it possible to construct scenes consisting of thousands of primitives within a fraction of a second.

Figure 7.4 shows the construction timings in relation to an increasing number of primitives (bicubic Bézier patches). Even with $100,000$ patches, the implementation requires only one second for the complete construction process. When distributing the construction work among two threads (dual-processor PC), a significant reduction of up to 50% can be achieved. Note that the multi-threaded timings still include thread creation time.

Applying ray-patch intersection that does not require an additional spatial index structure per-patch, e.g. an intersection based on uniform refine-

ment (see Chapter 6), allows for arbitrary dynamic manipulation by simply transforming the patch control points. In terms of complete Bézier scenes, one simply needs to reconstruct the kd-tree over the corresponding patches.

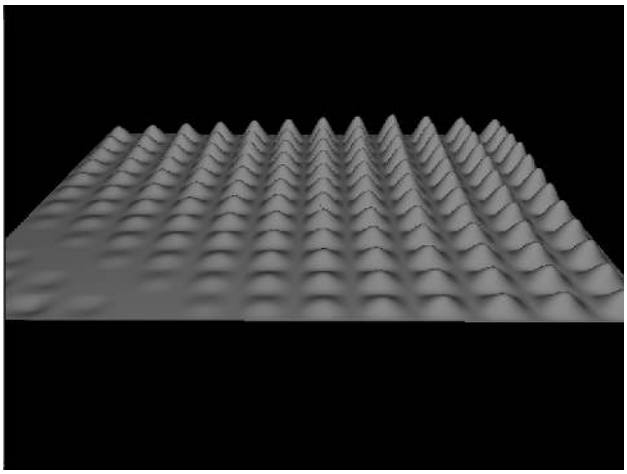


*Figure 7.5: Combining the tracing of ray bundles for Bézier patches with fast kd-tree construction allows for efficiently handling even completely dynamic scenes with arbitrary movement. This scene consists of* 144 *cones, each cone consisting of 4 Bézier patches (576 patches in total). The kd-tree over all patches is rebuilt from scratch for every frame. On a 2.2 GHz Pentium-IV, it runs with* 6.6 *fps at a resolution of* $800 \times 600$.

Figure 7.4 shows that even kd-tree reconstruction of $20,000$ patches requires less than 100 milliseconds. In order to demonstrate this new feature, a waving field (see Figure 7.5) of $12 \times 12 = 144$ cones was created, which is rebuilt from scratch for every frame. A build time of only a few milliseconds leaves enough time for ray tracing this fully dynamic scene with 6.6 fps on a 2.2 GHz Pentium-IV processor.

Apart from completely dynamic scenes, the startup time for highly complex scenes can also be significantly reduced. Figure 7.6 shows an artificial cube of $32,768$ teapots. Each teapot consists of 32 patches, corresponding to a total of $1,048,576$ patches. Even for this huge number of patches, a multi-threaded kd-tree construction makes it possible to achieve an almost immediate startup time of 4.42 seconds. Thanks to the ray bundle techniques of Chapter 6, this huge amount of patches can be rendered with 2.5 fps at a resolution of $800 \times 600$.

Applying the fast kd-tree construction algorithm to triangular scenes shows that the total rendering performance is reduced by about $20-40\%$ com-

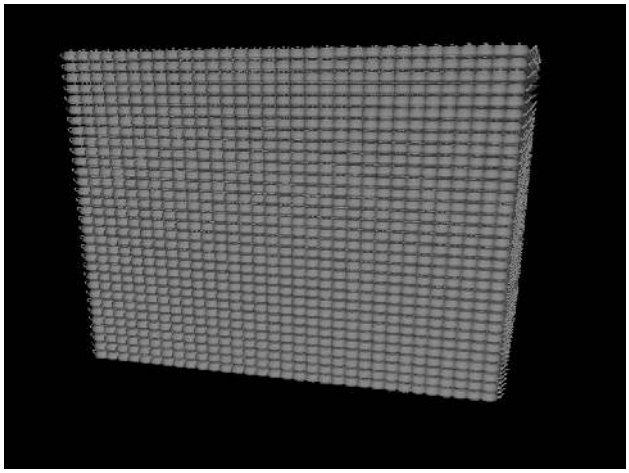


*Figure 7.6: With coherent ray bundles, even huge scenes of, for example,* $32,768$ *teapots (*$1,048,576$ *patches) can be interactively ray traced at* $2$ *fps at a resolution of* $800 \times 600$ *on a 2.2 GHz Pentium-IV. With multi-threaded kd-tree construction the startup time for this scene is reduced to less than* $5$ *seconds.*

pared to high-quality triangular-based kd-tree construction [Wald04]. However, compared to the construction time of the high-quality kd-tree, a 10 to 20 times faster construction is achieved, even when using only the single threaded version. As for Bézier patches, the fast kd-tree construction makes it possible to ray trace fully dynamic scenes of several thousand triangles.

Analyzing the fast kd-tree construction code using profiling tools such as VTune [Intel04] shows that for an increasing number of primitives (measured from $16,384$ to $1,040,576$), the sorting of split candidates takes up 30% to 35% of the total construction time.

As shown in Figure 7.4, the construction time is almost linear in the number of primitives. Usually, one would expect an increase in construction time if the working set becomes larger than the L2 cache size. However, even with hundreds of thousands of primitives, the major amount of the working set can be held in the CPU cache hierarchy. Moreover, the working set can be loaded very quickly into the cache hierarchy because the sequential access pattern of SAH evaluation and split sorting is very suitable for the built-in hardware memory prefetcher.

The actual bottlenecks of the code (measured by VTune) are located in the evaluation of the SAH and the sorting of splits according to the chosen split candidate. As a split is a compound data type (position and type),

a split comparison typically performs multiple branches. Therefore, split sorting involves a large number of mis-predicted branches. In terms of further optimization, one has to lower the amount of mis-predicted branches first. Removing branches by Boolean expressions might be a good solution to this problem. Evaluating the SAH for multiple split candidates in parallel using SIMD instructions could further reduce the number of mis-predicted branches and additionally increase the SAH split evaluation throughput. As a result, further low-level code optimizations could provide a significant speedup of performance.

## 7.2 Conclusions and Future Work

Multi-core CPUs will make it possible to efficiently construct kd-trees using multiple threads. The current implementation already supports two threads, where both of them are used for the split sorting step and the construction process. For the latter, the two threads start working independently once the split candidate list has been sorted according to the first chosen split candidate. In the case of multiple threads, the split candidate lists have to be sorted multiple times in order to provide a working set for each thread.

The time for generating the working sets might be further reduced by using either a hierarchical startup, where one thread starts two sub-threads, or an initial spatial sorting. A regular grid structure could be used as initial spatial sorting, permitting the sorting step to be performed very quickly. The number of non-empty grid cells would correspond to the initial number of working sets, which can be directly assigned to the set of construction threads.

Obviously, not all objects have to be reconstructed for every frame. If no ray accesses the object, the spatial index structure does not need to be rebuilt. In the case a ray accesses the bounding box of an object, a simple test could be performed if the index structure has already been built. Moreover, the rapid kd-tree construction algorithm could also be used for adaptive kd-tree construction [Havran01]. The idea behind adaptive kd-tree construction is to build the kd-tree on the fly during ray traversal. The more rays hit a given object, the finer the corresponding kd-tree must be built.

# Chapter 8

# Distributed Coherent Ray Tracing on Clusters

## 8.1 Introduction

Even though the compute power of a single or dual-processor workstation is quite sufficient for realizing interactive ray tracing with a small number of rays per pixel and simple shading, it is not enough for high frame rates in combination with many advanced shading effects.

In order to compensate for the need of compute power, researchers have used large shared-memory supercomputers (all processors share the same main memory) with up to hundreds of processors. Such a hardware setup has allowed for "pushing" ray tracing performance to realtime level rates [Keates95, Muuss95a, Muuss95b, Parker99b, Parker98, Parker99a], but the cost for a high-end supercomputer is tremendous. Because of the cost, these high-end systems are not very common.

On the "low-end" side, multi-processor systems [Intel01, AMD03, IBM05] with more than two processors (or cores) are not standard and therefore expensive. Even though this is beginning to change, arbitrarily increasing the number of processors within an off-the-shelf PC is often not possible. As the processor and the supporting hardware have to be exclusively designed for multi-processing, "low-end" multi-processor systems are typically restricted to a small number of CPUs. Unfortunately, the quality and complexity requirements of advanced ray tracing-based rendering algorithms can easily saturate the available compute power of small multi-processor systems.

Given the requirements of ray tracing, there is a tremendous need for an easy and inexpensive way to combine the compute power of several small single or multi-processor systems. In order to keep costs low, the method

of choice would be to connect the different systems by inexpensive off-the-shelf network technology. Unfortunately, this makes things more complicated because the total system would then be a distributed memory system (each desktop PC can only access its own main memory), and the interconnection framework for transferring data between the system's components suffers from a low bandwidth and a high latency. In order to compensate for this, a distribution framework is required that is able to handle the impacts of a distributed memory environment and high latency communication [Wald02a, Dietrich03, DeMarle03, Wald04, DeMarle04].

This chapter will provide a detailed presentation of the distribution framework that is used within the OpenRT library [Wald02a, Dietrich03, Wald04]. The framework has been exclusively designed for efficiently combining the compute power of a cluster of PCs, while handling all related memory and interconnection issues.

Before giving details about the OpenRT distribution framework, a brief overview of distribution strategies for parallel ray tracing will be presented first. Note that in the following a set of standard desktop PCs is referred as a cluster of rendering nodes.

## 8.2 Distribution Strategies

The main task for parallel ray tracing is to define a set of tasks that can be distributed across processors and executed in parallel. The preferable way for achieving the best performance is to split up the most time-consuming part of the rendering algorithm into independent tasks which can be executed in parallel. In terms of ray tracing, this typically means the shooting of rays.

In the following, a brief overview of typical tasks used for parallel ray tracing [Reinhard97, Reinhard95, Chalmers98, Chalmers01] is given:

**Frame Distribution:** In the context of off-line rendering, the tasks are typically defined as complete frames. Distributing frames across a render farm allows for achieving a high frame throughput, but the costs for rendering one frame are more or less constant.

**Scene Subdivision:** The scene data is partitioned by spatial subdivision. The resulting scene parts are distributed across the nodes of a distributed memory system. Based on the spatial location within the scene, the rays are transferred during traversal between the different system elements. This approach allows for rendering highly complex scenes that exceed the main memory of a single PC, since each system element handles only a fraction of the complete scene data. On the

other hand, rays must be reordered (based on their location) and all ray data has to be transferred across the network.

**Image Subdivision:** Due to the fact that the color of each pixel can be computed independently of any other pixel, a distribution strategy based on image plane subdivision can be used to speedup the rendering of a single frame. In order to display the final frame, only the computed pixel colors need to be combined.

Looking at the distribution schemes in detail makes it clear that not all of them are suitable for realtime ray tracing on a distributed memory system. A frame-based distribution would introduce a latency corresponding to the amount of time required for rendering all distributed frames. For a realtime system requiring immediate feedback, the approach is not acceptable.

On the other hand, scene subdivision is also sub-optimal because the limited bandwidth of the interconnection framework makes it difficult to transfer data quickly. As rays typically traverse large spatial regions, many different spatial locations are accessed. Therefore, a huge amount of ray data must be transferred between nodes. Representing a ray usually requires at least 32 bytes (direction, origin, intersection distance, primitive id) and assuming a resolution of $1024 \times 1024$ pixels and 10 rays per pixel, yields 320 MB per frame. This amount of data simply exceeds the bandwidth of today's interconnection techniques.

Additionally, the connection of spatial locations to nodes typically creates "hot spots" during ray traversal, because most of the time only a few nodes actually work in parallel. For example, the nodes containing the camera or the light sources will need to handle numerous rays, thereby becoming "hot spots". Due to this sub-optimal utilization and the consequently limited scalability, the OpenRT framework instead uses image plane subdivision for task distribution.

A typical image-based distribution approach relies on individual pixels. For example pixel $i$ would be assigned to node $i \ mod \ n$, where $n$ is the number of nodes. Even though the required transfer of pixel coordinates and colors would require less bandwidth than sending ray data, the amount is still significant. Assuming two bytes per pixel coordinate yields 4 MB, without the actual pixel colors for a resolution of $1024 \times 1024$ pixels. Additionally, relying on individual pixels complicates the gathering of coherent primary ray bundles because of the spacing between pixels assigned to a particular node.

Given bandwidth requirements, a distribution scheme with an image plane subdivided into rectangular tiles is more beneficial. Rectangular tiles

can be easily represented by only two 2D pixel coordinates, thereby largely reducing the amount of transferred data. Assuming a tile size of $32 \times 32$ results in 1024 tiles and yields 4 KB of coordinate data per frame (without pixel colors). This amount of data is acceptable for transferring data across the different nodes.

The image-based distribution scheme makes it necessary to provide each node either with direct access to the complete scene data or enable it to load required data on demand [Wald01a, DeMarle03].

Besides the actual definition of the tasks, the distribution scheme itself is of high importance. Assigning the pixel tiles for a frame statically to the nodes in the same manner as individual pixels, works well if the required rendering time per tile is roughly the same. In this case, all nodes finish the rendering of a frame at the same time, fully exploiting parallel execution. If the rendering time per tile varies significantly, however, a statical distribution will increase the time necessary for rendering a frame because of the low parallel execution at the frame's end. Note that the smaller an image tile is, the more likely equal rendering times are because the rendering load will be more finely distributed.

Given the bandwidth requirements, image tiles have to be rather large compared to the size of individual pixels. Therefore, the rendering time per tile is likely to vary significantly, making a statical distribution scheme sub-optimal. Therefore, the OpenRT distribution framework distributes the image tiles dynamically to nodes, to ensure full utilization of all nodes.

In the following, the OpenRT distribution framework will be discussed in more detail.

## 8.3 The OpenRT Distribution Framework

The OpenRT distribution framework [Wald02a, Dietrich03, Wald04] is based on a master/slave(s) approach. The master is responsible for task and scene data distribution, while the slaves do the actual rendering work. The master then receives the pixel colors of image tiles rendered by the slaves and combines them into the final image.

Physically, a master or slave is just a PC within the cluster. As discussed in Section 8.2, the master uses a tile-based task distribution, which means that the master sends the definition for a rectangular tile to one of the slaves, waits until the slave has rendered all pixels within this particular tile, and finally receives the corresponding color data. Once the master has received all color data, it displays the frame.

The master is also responsible for distributing the scene data to all slaves.

The same holds true for scene data updates, e.g. a new camera or light source position, which are once more transferred from the master to all slaves. Note that this transfer is unidirectional, as there is no need for transferring scene data from slave to master. For the OpenRT framework, there is no difference between the initial scene data distribution and subsequent scene data updates, because an application communicates with the underlying library only through the OpenRT API. The OpenRT framework distributes each API call, with the exception of geometry calls, to all slaves. This mechanism is similar to remote procedure calls, except that OpenRT distributes the API calls asynchronously. Geometry calls are cached on the master until the corresponding object is completely defined. Geometry caching avoids the distribution of many geometry calls, saving costly bandwidth.

The main issue of the tile-based task distribution is to avoid idle periods on the slave side. These idle periods can have the following reasons:

**Communication Latency:** When relying on standard network technology, the network round-trip time (master-slave-master) is in the region of milliseconds. During such a time span, one could trace thousands of rays, which means that communication-related dependency chains between master and slave should be avoided or kept to a minimum. If, for example, the master only sends a new task to a slave after having first received the color data of a previous tile, the slave would spend a lot of time waiting for the next task.

**Synchronization:** Parallel execution is always affected by synchronization, because synchronization means serialization. In terms of an image-based ray tracing system, synchronization occurs after the end of frame $n$ and before the start of frame $n+1$. Only during this period can the scene data be updated (during rendering, it has to be fixed to avoid inconsistent scene data).

The OpenRT framework minimizes these idle periods by introducing asynchronous behavior and special latency hiding techniques:

**Dynamic Load Balancing:** The scene and shading complexity typically vary across the image plane. Therefore, a dynamic load balancing scheme is used. Each slave requests a task (tile) from the master. If frequently the rendering of tiles takes longer, the corresponding slave will ask for work less often, allowing other slaves to take over part of the image computation. This ensures that a high variation in rendering time complexity will be handled more efficiently.

**Task Prefetching:** In order to hide communication latency on the slave side when requesting tiles from the master, a fixed number of tiles is assigned to each slave in advance. These tiles are stored on the slave side in a task queue. By transferring the color data of a computed tile back to the master, the slave simultaneously requests a new tile from the master. However, as there still are tasks left in the task queue, the slave can directly continue with the next task without having to wait for the arrival of the new one. In order to prevent stalls, the time required to render the tiles within the task queue must be greater than the round-trip time for sending and receiving a tile.

**Asynchronous Data Transfer:** Given the limited bandwidth and high latency, a transfer of data across the network is costly. The most critical time span is required for transferring scene data updates from the master to the slaves because the master has to broadcast the data. Therefore, asynchronous data transfer has been integrated: While slaves are rendering frame $n$, the master directly transfers the scene data for frame $n+1$ to the slaves. During rendering of frame $n$, frame $n-1$ is displayed. This introduces one frame of latency between the rendering and the display of a frame, but gives the master the time span of one frame to transfer scene data updates to all slaves.

**Two-Frames Task Scheduling:** Even though dynamic load balancing tries to distribute the rendering task evenly over the slaves, idle periods on the slaves occur when the master has to synchronize all slaves at the end of a frame. In this case, all tiles of a given frame have been sent, but the corresponding rendering has not been finished yet on the slave side. Before the master can distribute the tiles of the next frame, it has to wait until all tiles from the previous frame have been received. In order to prevent idle periods because of frame synchronization, the master allows the slaves to continue rendering with the next frame (even if the current frame has not yet been completely finished by all slaves). In the event a slave has already received the scene data for the next frame, it can directly proceed with the rendering (if not, the slave has to wait until data reception is completed).

Supporting latency hiding techniques requires asynchronous behavior on the master and the slave side. In the following sections, the master and slave implementation used in the OpenRT framework will be presented.

The implementation has been exclusively designed for high-performance. Therefore, no sophisticated library such as MPI [Foruma], PVM [Geist94] was used to prevent abstraction layer overhead in network communication.

Instead, the implementation uses a framework that is directly based on the standard UNIX sockets [Stevens98]. All communication relies on the TCP/IP protocol. Even though communication based on UDP would offer features such as network broadcast, the possible loss of data would require special and complex countermeasures. For parallel thread execution, OpenRT relies on the POSIX pthread library [Nichols96].

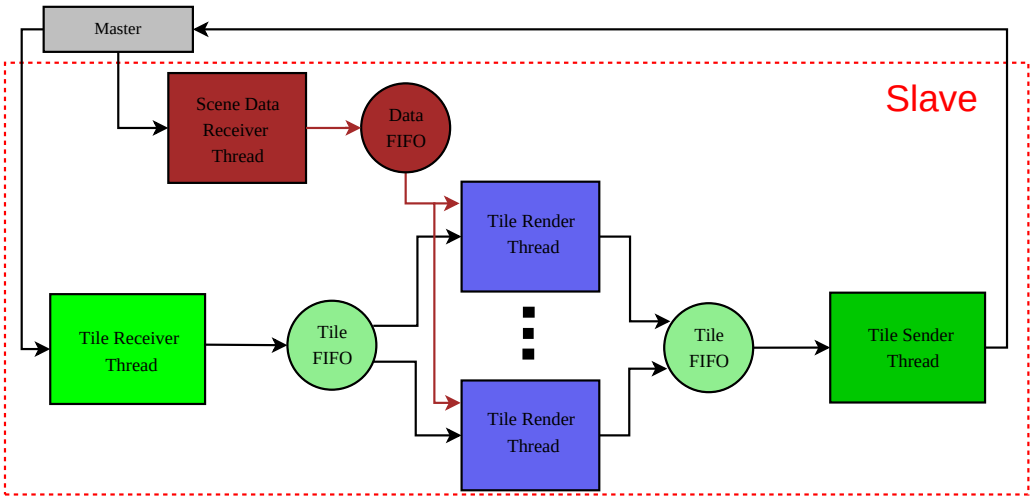



*Figure 8.1: Multi-threaded slave implementation: Each colored box corresponds to a single thread, a colored circle representing a shared data structure. The green and blue rectangles form the rendering pipeline, which receives image tile data over the network from the master, performs the rendering and sends the corresponding pixel colors back to the master. The second pipeline which runs asynchronously to the first, receives scene data from the master over the network, and stores it in the data queue. Once all tiles for the given frame have been rendered, the rendering threads are synchronized and the scene data is updated based on the content of this queue.*

### 8.3.1 Slave

The slave relies on two asynchronously running pipelines, each consisting of one or multiple threads (see Figure 8.1). The first pipeline, the so-called *rendering pipeline*, is responsible for receiving image tiles from the master, performing the actual rendering, and for sending back the color data to the master. Each of these three steps is handled by a separate thread.

The "execution flow" for the rendering pipeline is as follows: The *Tile Receiver Thread* is listening on a network socket connected to the master. As

soon as the *Tile Receiver Thread* receives an image tile, it puts the data into a *Tile FIFO Queue*. A number of *Tile Render Threads* takes entries from the queue and performs the actual rendering. The number of *Tile Render Threads* is usually related to the number of CPU cores on the slaves. After rendering, each *Tile Render Thread* adds the pixel colors to the image tile and inserts the combined data into the second *Tile FIFO Queue*. The last thread, the *Tile Sender Thread*, takes a tile (with pixel colors) from the last queue and sends all corresponding data (over a network socket) back to the master.

The second pipeline, the so-called *scene data pipeline*, is responsible for handling the scene data updates. Therefore, the *Scene Data Receiver Thread* listens on the second incoming network connection to the master. Through this connection, the master sends scene data updates, which will be stored in the *Scene Data Queue*. Buffering of received scene data is necessary because a scene data update must be synchronized with the *Tile Render Threads*. Once a frame has been completely rendered, the set of *Tile Render Threads* is synchronized, and a scene data update can be safely performed based on the content of the *Scene Data Queue* before restarting the *Tile Render Threads*.

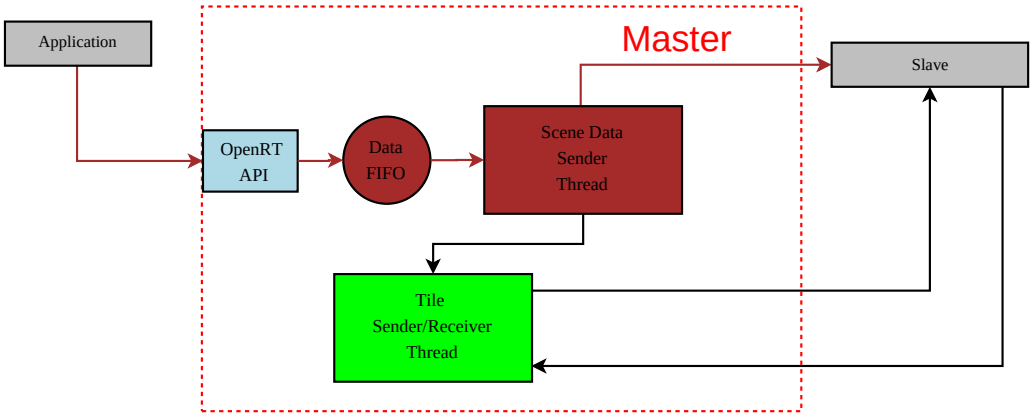



*Figure 8.2: Multi-threaded master implementation: Each colored box corresponds to a single thread. The scene data queue, together with a thread for sending scene data to the slaves, builds the application pipeline. This pipeline runs synchronously to the application. The second thread, which forms the slave pipeline, runs asynchronously with the application. While the application defines the scene for frame n, the application pipeline buffers the corresponding data, while the same pipeline asynchronously handles the image tile distribution and reception for the two previous frames.*

### 8.3.2 Master

Similar to the slave, the master consists of two asynchronously running pipelines (see Figure 8.2). The first pipeline, the so-called *application pipeline*, runs synchronously with the application, whereas the second pipeline, the *slave pipeline*, runs asynchronously to the application.

The main task of the *application pipeline* is to distribute scene data updates (in the form of OpenRT API calls) over the network to all slaves. The point in time when all scene data for the current frame is defined, is also the point of synchronization with the second pipeline. At the synchronization point, the application pipeline waits until all pixel data for a new frame has been received and completely copied to the frame buffer.

The slave pipeline consists of a single thread, called *Tile Sender/Receiver Thread*, which handles all tile-based communication between master and slaves. It listens on the network sockets which connect the master to the slaves. It responds to each incoming tile color data with a new tile task. If all tile tasks for the current frame have already been assigned, tiles for the next frame are used. Therefore, tiles for a maximum of two frames can be in-flight. As a result, the *Tile Sender/Receiver Thread* must be capable of handling tiles for two different frames at the same time, which directly translates to supporting double-buffering via two internal frame buffers.

## 8.4 Communication and Dataflow

For a better understanding of the asynchronous behavior of master and slave, a simplified timing diagram is shown in Figure 8.3.

While the master receives all tile data for frame $n$, frame $n-1$ is displayed. The master's application pipeline is locked until all tiles for frame $n$ have been received. At this point, the lock for the application pipeline is released, and frame $n$ can be displayed by the application. Moreover, the application can already start to specify the scene for frame $n+1$.

Due to an one-to-one exchange of color data and new tasks, the master has to distribute tiles for frame $n+1$ (all tiles for frame $n$ have been distributed), while still waiting for the color data of frame $n$. This is the key point to prevent idle periods on the slaves, because the master allows those slaves to proceed with the rendering of frame $n+1$ which have already received the scene data for frame $n+1$.

The slave receives, renders, and sends back tile data for frame $n$, while asynchronously receiving the data for frame $n+1$. When all scene data for frame $n+1$ has been received, the slave updates the scene data by the

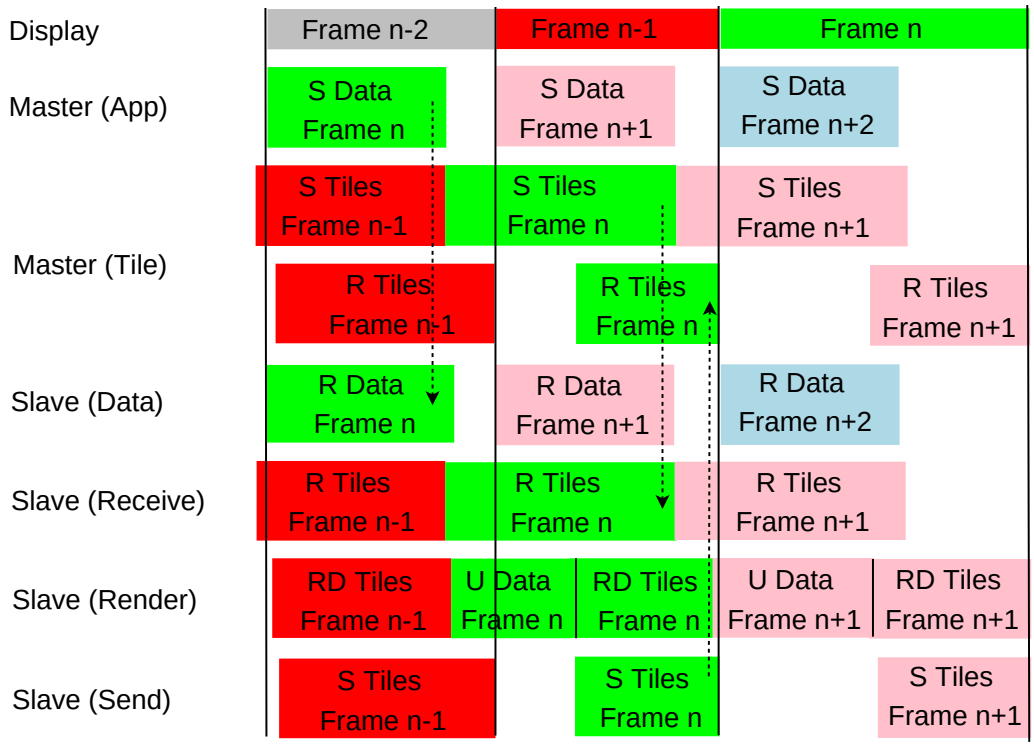



*Figure 8.3: The timing diagram for master-slave communication (S = Send, R = Receive, U = Update, RD = Render). The main synchronization point appears when all tiles for frame n have been received. Only at this point is the application able to display frame n. While distributing tile jobs and receiving color data back for frame n, the master sends data for frame n + 1 to all slaves. The slave asynchronously buffers this data, while it receives, renders, and sends data for frame n. Therefore, one frame of latency between the specification and display of a frame is introduced.*

buffered data, while the master is still waiting for color data of the previous frame. As long as all scene data is received before the rendering of the last image tile is finished, the time for the network transfer can be completely hidden. If the time required for updating the scene data is sufficiently small, the slave will have almost no idle periods in terms of rendering. Between the specification and the display of a frame, one frame of latency is introduced.

In the case of using multiple rendering threads, the update must only be executed by a single thread. Therefore, all rendering threads need to be serialized at this point.

As discussed in Section 8.3, the master distributes a set of tiles in advance.

These tiles are directly buffered by a *Tile FIFO Queue*, so that they can be directly requested by a rendering thread. If the master receives the color data for the first tile of the set, it will send a new tile. In the case of $n$ rendering threads, the time required for rendering all tiles in the queue decreases to $1/n$ of the original time span. Therefore, the number of tiles sent in advance must be increased accordingly.

## 8.5 Results

For testing the scalability of the master/slave distribution framework, a cluster of 24 dual Athlon MP 1800+ (48 CPUs in total) with 512 MB memory was used. For each CPU, a separate slave process was created, yielding a maximum of 48 slaves running simultaneously.

Since all slaves transfer the image data to the master, the network bandwidth to the master can easily become a bottleneck. Therefore, the slave PCs are interconnected by a fully switched 100Mbit Ethernet using a Gigabit uplink to the master.

A set of four triangular example scenes was rendered at a resolution of $640 \times 480$, using only primary rays and simple shading. The scene complexity ranged from a few thousand triangles for the "Office" scene up to several millions for the "Power Plant" scene. For a detailed scene description see [Wald03e, Wald04].

Figure 8.4 shows that for all example scenes the distribution framework achieves linear scalability in the number of CPUs (slaves) until the network bandwidth to the master is saturated. For the given setup, the saturation appears when more than 22 MB/s have to be transferred over the network. For a resolution of $640 \times 480$ and color coding of three bytes per pixel, this corresponds to a maximum of 25 frames per second. By combining the bandwidths of several Gigabit network connections, the framework even allows for reaching $40 - 50$ fps.

## 8.6 Conclusions and Future Work

The proposed master/slave-based distribution framework allows for efficiently combining the compute power of several PCs interconnected by standard network technology. The framework has been designed especially for scalability and interactivity, while relying on a high-latency interconnection. In order to hide the interconnection latency, several techniques such as dynamic load-balancing, task scheduling, task prefetching and asynchronous data transfer

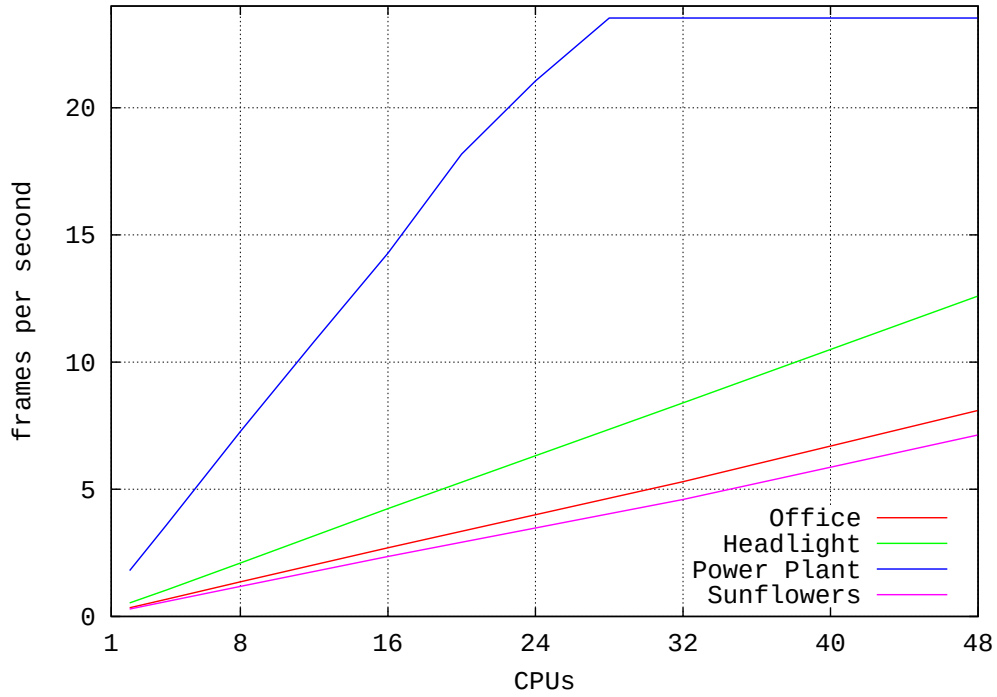



*Figure 8.4: Scalability of the master/slave distribution framework for an increasing number of CPUs (slaves) and different example scenes. All scenes have been rendered at a resolution of* $640 \times 480$ *using only primary rays and simple shading. The complexity of scenes ranges from a few thousand triangles for the Office scene up to several million triangles for the Power Plant scene. For all scenes, the distribution framework achieves linear scalability in the slaves until the network bandwidth to the master is saturated. The saturation appears for the given hardware setup at a maximum transfer rate of 22 MB/s, which corresponds to 25 fps.*

have been integrated into the framework. These techniques allow for achieving a linear scalability in the number of connected PCs.

Even though the distribution framework provides linear scalability, the maximum frame rate is limited by the maximum network bandwidth. For the network setup shown in Section 8.5, a maximum of 25 frames per second can be achieved. For future setups, it will be necessary to further increase the bandwidth to the master. Obviously, faster network technology like Myrinet [Forumb] or Infiniband [Futral01], could provide a higher bandwidth but the required network hardware is still non-standard and thus expensive. Therefore, it might be beneficial to rather focus on fast compression

and decompression algorithms in order to reduce the amount of transferred data. For multi-core CPUs, in particular, one could reserve a single core for compressing/decompressing pixel data.

In the current implementation, specific system parameters such as the size of a pixel tile or the numbers of tiles that a slave fetches in advance are fixed. This typically works well for most situations, while it can be sub-optimal in others. Therefore, future implementations should automatically analyze the efficiency of the applied parameters by, for example, measuring the load on the slaves and, if required, adapting these parameters during run-time to ensure an optimal work distribution.

# Chapter 9
# Applications

A key point for achieving high ray tracing performance is ray coherence. As discussed in Section 4.4.5, most of the ray tracing-based rendering algorithms, e.g. standard backward ray tracing, rely on the recursive evaluation of single ray paths. The recursive evaluation makes it difficult to combine individual rays to ray bundles. Even if individual rays can be combined, it is not ensured that the rays within the bundle are coherent. Unfortunately, this also applies for any other ray-path generating rendering algorithm.

Solving this problem requires a shift in design: Instead of trying to gather coherent rays out of an arbitrary rendering algorithm, one should design, or modify, the rendering algorithm in order to produce coherent rays.

In the following section, a rendering algorithm is presented that is able to almost exclusively produce coherent ray bundles. This rendering algorithm is known as the *instant global illumination algorithm*, and was originally presented by Wald et al. [Wald02b, Wald04]. By having a fast ray tracing core combined with a lot of processing power, the algorithm is able to fully recompute a complete global illumination for every frame. The improved algorithm [Benthin03] described in this chapter has been designed to fully exploit coherent ray bundles and to eliminate the bottlenecks of the previous algorithm [Benthin03].

## 9.1 Instant Global Illumination

This section briefly reviews the original instant global illumination algorithm. For a more detailed description of this algorithm and other global illumination algorithms see [Wald04].

The core of the instant global illumination algorithm is build on the *instant radiosity algorithm* proposed by Keller et al. [Keller97]. Instant radios-

ity approximates diffuse illumination by a set of virtual point light sources (*VPLs*). These VPLs are computed in a preprocessing step by generating light paths (using random walks) from the light sources into the scene. At each light path bounce, a VPL is created (the emittance of the virtual point corresponds to the reflected radiosity at the given point). The contribution of all VPLs approximates the indirect illumination.

The set of VPLs is used in the actual rendering process to compute the irradiance at every point. For each primary ray intersection point, the irradiance is computed by shooting shadow rays towards all generated VPLs and adding their contributions. Obviously, the more VPLs are considered per intersection point, the more accurate the illumination computation will be. However, as the focus is on interactivity, only a small set of VPLs is affordable per pixel.

VPLs cause discretization errors due to the generation of sharp shadow boundaries. However, these can be smoothed by using *interleaved sampling* [Keller01]. The idea behind interleaved sampling is to not generate and use only one set of VPLs for all pixels but to generate multiple VPL sets and interleave these sets for the illumination computation of neighboring pixels. An $N \times N$ pixel pattern implies the generation of $N \times N$ VPL sets. As neighboring pixels do not share the same VPL set, hard shadow boundaries are broken up. All VPL sets are of equal size and, therefore, the number of shadow rays per pixel is similar to the one-set case. This implies that no additional work has to be done. The drawback of interleaved sampling is the transfer of discretization errors into visible structured noise. The combination of interleaved sampling and instant radiosity is illustrated in Figure 9.1.

The structured noise introduced by interleaved sampling can be significantly reduced by a technique which is called *discontinuity buffering* [Keller01, Wald02b, Wald04]. Discontinuity buffering essentially filters the irradiance of neighboring pixels. In order to filter over all different VPL sets, the filter kernel must have the same size ($N \times N$) as the interleaved sampling pattern. Discontinuities, e.g. surface edges or cured surfaces must not be filtered, otherwise visible blurring artifacts will appear. Therefore, discontinuity buffering relies on several heuristics (hit point distance, surface normal) to detect such cases. The key point of discontinuity buffering is that it allows for combining the illumination computation of neighboring pixels. For an $N \times N$ interleaved sampling pattern and an $N \times N$ filter kernel, every pixel exploits the illumination from all $N \times N$ VPL sets. The combination of interleaved sampling and discontinuity buffering trades spatial blurring of the illumination for the speed of computation. Usually, the results are acceptable because indirect illumination changes only slowly on surfaces.

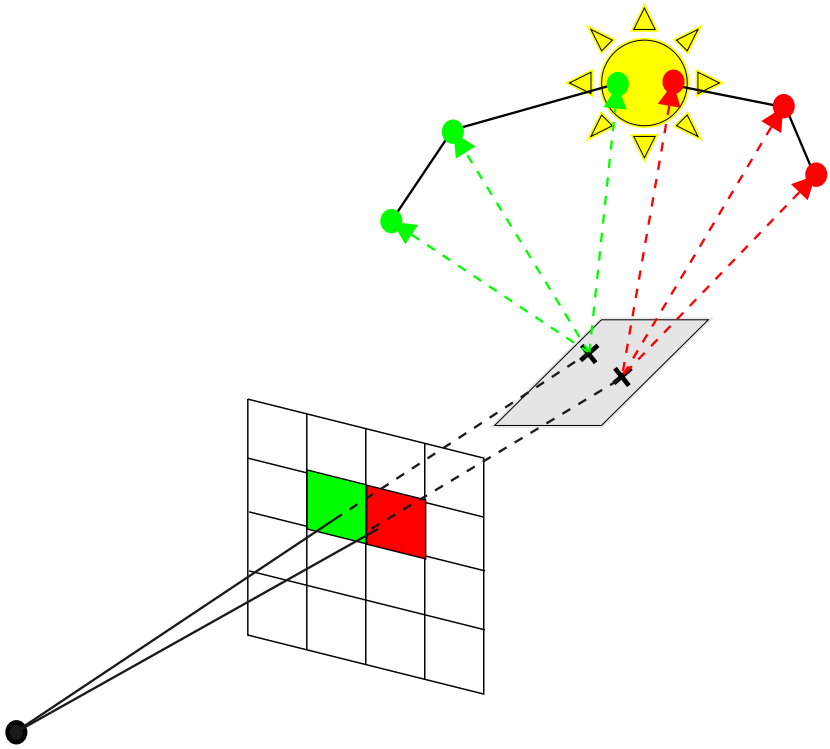


*Figure 9.1: Combination of instant radiosity and interleaved sampling: Neighboring pixels use a different set of virtual point lights (VPLs). The structured noise introduced by the different VPL sets is on removed later on by filtering the irradiance of neighboring pixels.*

## 9.2 Exploiting Coherence

The standard approach of combining rays from neighboring pixels to coherent (shadow) ray bundles as shown in Chapter 4 does not work any longer when using interleaved sampling. Therefore, only pixels that share a common VPL set produce shadow rays which are coherent. As a result, only those rays should be combined into bundles. One could trace primary rays from neighboring pixels together and later reorder the intersection points according to the different VPL sets. However, reordering would require many scatter-gather operations, which are costly when using SSE instructions.

Fortunately, reordering is not necessary because an $N \times N$ interleaved sampling pattern ensures that every $N^{th}$ pixel (with respect to both rows and columns) has the same interleaved sampling index $i, 0 \leq i \leq N \times N$. If $N$ is small, e.g. 3, grouping primary rays according to their interleaved sampling

index will only slightly decrease coherence within the bundle. The increased space between primary rays for a $3 \times 3$ interleaved sampling pattern causes an overhead of approximately 10% to 20% for primary ray bundles when rendering at moderate resolutions, e.g. $640 \times 480$. For higher resolutions, e.g. $1024 \times 1024$, the overhead is clearly below 10%.

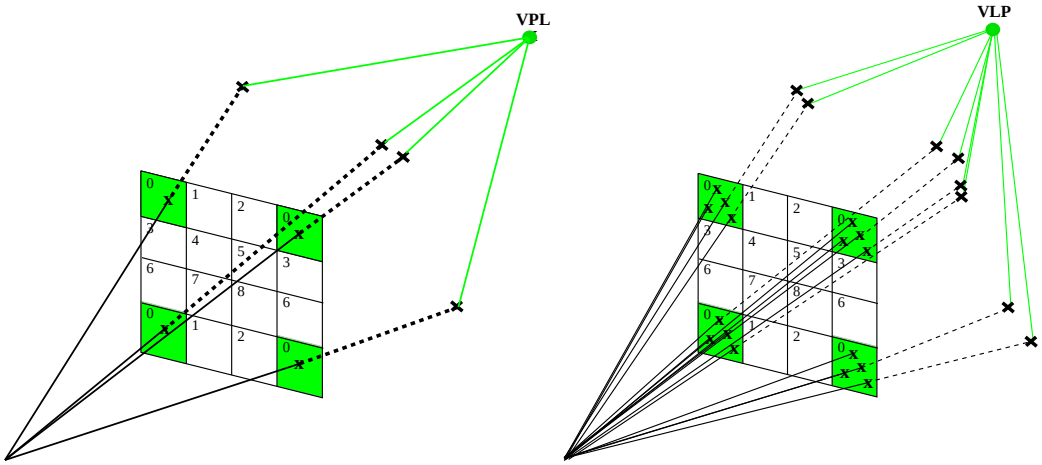



*Figure 9.2: Combining primary and shadow rays which share the same interleaved sampling index (marked green) into bundles. A $3 \times 3$ interleaved sampling pattern ensures that coherence within a bundle is only slightly reduced. Left Image: Casting one ray per pixel, four rays per bundle in total. Right Image: Casting four rays per pixel, sixteen rays per bundle in total. This arrangement is suitable for per-pixel super-sampling.*

Chapter 4 showed optimized routines for bundle traversal of 4 or 16 rays. These core routines can be directly reused for tracing primary and shadow ray bundles which share the same interleaved sampling index. Figure 9.2 illustrates these two approaches: The first casts only one ray per-pixel, the second casts four rays per pixel for improved anti-aliasing by per-pixel super-sampling. In terms of improved per-pixel anti-aliasing, different approaches for gathering coherent rays exist, e.g. casting bundles of 4 or 16 rays per pixel without adding rays from pixels sharing the same interleaved sampling index. These approaches offer higher coherence but significantly more rays have to be shot (16 rays per-pixel). As the focus lies on interactivity, Section 9.4 will present a way for improved per-pixel anti-aliasing without introducing significant performance penalties.

## 9.3 Streaming Computations

In order to fully exploit the benefits of SSE instructions, the entire process of ray shooting and shading operations is reformulated in a streaming-like and breadth-first way. Instead of directly proceeding to shading and shadow ray generation after having cast the first primary ray bundle, multiple neighboring primary ray bundles with the same interleaved sampling index are cast first. The results, e.g. intersection point, surface normal, etc. are stored. Based on the stored intersection data, the process continues with the generation and casting of shadow rays to each VPL in the current set. The contribution of each VPL is added to the stored intersection data. After adding the contribution for the last VPL in the current set, the process continues with shading evaluation and the casting of additional secondary rays (see Section 9.3.1).

This breadth-first approach significantly increases cache usage because neighboring ray bundles are likely to access similar data.

### 9.3.1 Streaming Shading

As the cost for shading can easily exceed the cost for ray casting, a highly efficient shading system is essential. However, allowing for freely programmable shaders aggravates the problem of realizing efficient shading.

As a solution, the actual shading process is split into the evaluation of two independent entities: a fixed function *BRDF shader* and a freely programmable *surface shader*. The BRDF shader evaluates the illumination contribution and the underlying BRDF itself. The key point is that the BRDF shader performs the evaluation based on the settings provided by the surface shader. The surface shader itself only provides shading parameters, e.g. diffuse, specular color, normals, etc. and passes them on to the BRDF shader which is responsible for tracing secondary rays.

This programming model allows for maintaining high shading performance thanks to a highly optimized BRDF shader, while providing most of the features of programmable shading. The high performance is accomplished by shifting the critical performance part, e.g. adding the illumination of VPLs and BRDF evaluation, to a highly optimized BRDF shader. As bundles of rays return bundles of intersection points, the BRDF shading must also be performed for bundles. This offers the advantage that code dependencies between shading computations can be minimized, allowing for increased instruction level parallelism.

The usually less critical shading part is taken on by a freely programmable surface shader. If costs for tracing rays dominate the total rendering cost,

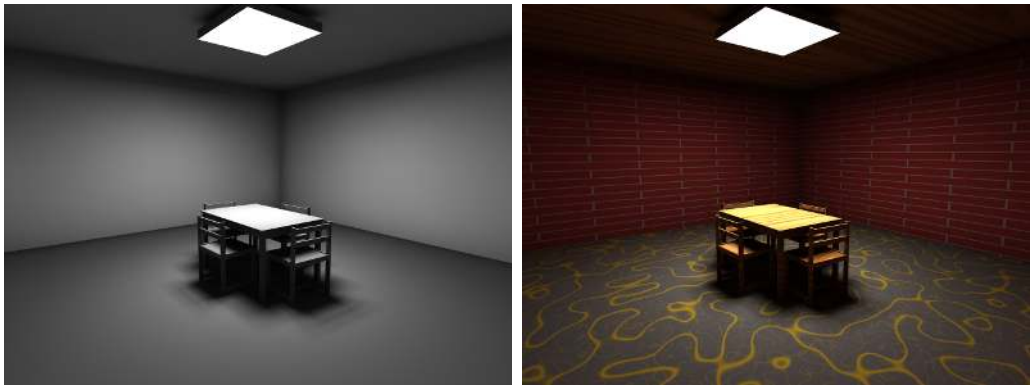


*Figure 9.3: Freely programmable procedural shading in a globally illuminated scene. The "Shirley 6" test scene (left) in its original form, and after applying several procedural shaders (marble, wood, and brick-bump). The efficient shading model shifts the performance bottleneck towards ray casting. Therefore, even complex and costly shading operations, such as multiple noise function evaluations, only moderately affect total rendering performance. In this case, performance drops to* 3.7 *fps compared to* 4.5 *fps with constant diffuse reflection.*

the relative costs for the freely programmable part are relatively low. This allows for complex shading operations without large performance penalties (see Figure 9.3).

For standard shading, profiling shows that almost all shading time is taken up by the BRDF shader. This especially applies for adding up the weighted illumination of VPLs: Each VPL in a scene is described by its position $x_i$, the surface normal $n_i(x_i)$, and its emitted radiance $L_i$. The irradiance at any given surface point $x$ with normal $n$ is then approximated by adding up the contributions from all VPLs:

$$E(x) = \sum_{i=0}^{N-1} V(x, x_i) \frac{\cos\theta_x \cos\theta_i}{||x - x_i||^2} L_i, \tag{9.1}$$

where $V(x, y)$ specifies the visibility between two points $x$ and $y$, and $\theta_x, \theta_i$ are the angles between the normal vectors $n_x$, $n_i$ and vector $x - x_i$, respectively. Based on the normalized surface and VPL normals, the two cosine factors are evaluated by two dot products, allowing for an efficient SSE implementation. Note that $E(x)$ is evaluated for multiple intersection points in parallel which allows for efficiently using SSE instructions to perform the dot products. As VPL normals are normalized before the rendering starts, the normalization

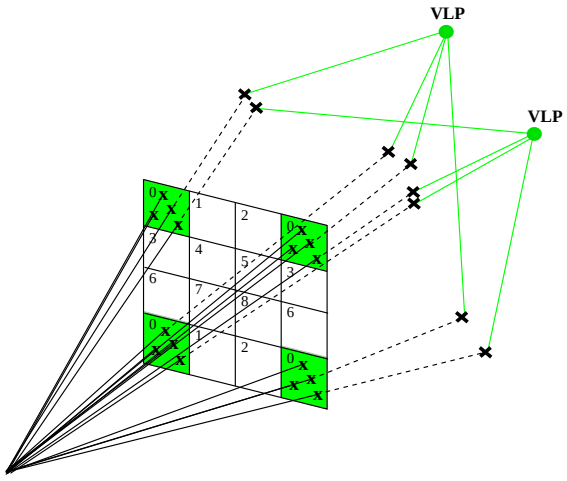



*Figure 9.4: Efficient anti-aliasing using a similar interleaving approach as for the generation of VPLs. Instead of connecting all intersection points of a given pixel to the same VPLs, the intersection points are connected to different subsets of the same VPL set.*

only needs to be performed for surface normals. Each surface normal is reused for evaluating $E(x)$ for each VPL. This allows for efficiently amortizing the normalization cost.

Evaluating $V(x, y)$ requires casting shadow rays. If the surface and VPL normals indicate that no radiance transfer can occur, the shadow ray casting could be skipped. However, as shadow rays are cast in bundles, the skip decision has to be evaluated with respect to the entire bundle. The related decision coherence is very high, because the surface normal usually does not vary significantly for adjacent spatial locations.

## 9.4 Efficient Anti-Aliasing

Efficient per-pixel anti-aliasing can be realized for the instant global illumination algorithm by following a similar interleaving approach as for the generation of the different VPL sets (see Figure 9.4). Instead of connecting each one of $N$ primary ray intersection points of a given pixel to all VPLs, the VPL set is split into $N$ sub-sets, and each intersection point is only connected to the VPLs from one sub-set.

The main advantage of this approach is that the total number of rays per pixel is only slightly increased. The only "investment" is the increased

number of primary rays per pixel while the number of shadow rays remains constant. Depending on the size of the VPL set, the performance impact of efficient anti-aliasing typically lies in the range of 10% to 30%, while the actual image quality is significantly enhanced (see Figure 9.5). Different primary rays for the same pixel obtain illumination from different VPLs; a possible difference in the illumination is therefore possible (compared to connecting each primary ray to the same VPL set). In most cases, however, each primary ray is connected to multiple VPLs, thereby reducing the probability of illumination discrepancies. For typical scenes, the difference in illumination quality is virtually indistinguishable.

*Figure 9.5: Left: Casting a single primary ray per pixel exhibits aliasing artifacts for small geometric details. Right: Casting four primary rays per pixel with interleaved shadow ray generation. As the number of shadow rays is constant, performance decreases only from 4 fps to 3.2 fps while providing better image quality through anti-aliased rendering.*

## 9.5 Distributed Rendering

High image quality requires many VPLs which in turn requires casting of many shadow rays. In order to perform a full recomputation of the complete illumination per frame, the compute power of a single PC is not sufficient. As a result, multiple PCs need to be combined. The original instant global illumination system [Wald02b] as described here therefore uses the distribution framework presented in Section 8.3.

The original instant global illumination system [Wald02b] performs the rendering on the slaves and the discontinuity filtering on the master. As a

result, all pixel data required for discontinuity filtering needs to be transferred to the master. The required bandwidth easily saturates the network bandwidth, thereby limiting the scalability of the entire system. The support of caustics in the original system makes it necessary to perform the filtering on the master side. By dropping caustics support in the new system, the filtering step can be moved to the slave side.

A drawback of filtering on the slave side is the overhead caused by the tile-based task distribution system. More precisely, a $3 \times 3$ filtering kernel requires access to pixel data which lies outside the tile boundaries. Therefore, the tile has to be virtually enlarged by a one-pixel border. Obviously, the larger the tile, the smaller the overhead will be. For a tile size of $40 \times 40$ pixels, 164 additional pixels have to be computed. This 10% overhead is easily acceptable when gaining better scalability in return.

An advantage of filtering on the slave side is that all filtering data can be completely held in CPU caches. This is beneficial because the filtering step is then completely memory-bound. As the filter kernel has a regular and fixed size and all input values are provided as single precision floating point values, the actual filtering is efficiently implemented using SSE instructions. If all input values are directly loaded from cache, the SSE-based filter easily outperforms a non-SSE version by a factor of about 3.

## 9.6 Results

The new instant global illumination system [Benthin03] demonstrates the effectiveness of directly supporting coherent ray bundles using rendering algorithms. Streaming shading and filtering (see Section 9.3.1) shift the critical computational part towards ray casting. Combined with the typical setting of using many VPLs per pixel, the total rendering time is strongly dominated by ray casting. This allows for directly exploiting the fast traversal code for ray bundles from Chapter 4.

Table 9.1 shows a comparison between the original and the new instant global illumination system. The original system only supports single ray traversal and no streaming shading. The new system's support for tracing bundles of rays and streaming shading allows for outperforming the old system by a factor of $3 - 8$. The higher the resolution, the higher coherence within the ray bundles and the more beneficial bundle traversal becomes. Note that the statistics for the new system already include the 10% overhead caused by overlapping at tile boundaries.

Even the compute power of a single PC is sufficient to achieve interactive frame rates at small resolutions. Table 9.2 shows the actual performance per

| | Office | Conference | Power Plant |
|---|---|---|---|
| $640 \times 480$ | 2.7 | 3.2 | 2.5 |
| $800 \times 600$ | 2.9 | 3.41 | 2.7 |
| $1000 \times 1000$ | 4.2 | 7.2 | 4.0 |
| $1600 \times 1200$ | 5.7 | 8.0 | 5.1 |

*Table 9.1: Performance increase of the new compared to the previous instant global illumination system using 64 VPLs/pixel with full shading and filtering at various resolutions. Even with a 10% overhead caused by overlapping tile boundaries, the new system outperforms the original one by a factor of up to 8 at higher resolutions.*

CPU in million rays per second. Note that these numbers include shading and filtering. The table further demonstrates the impact of coherence. Increasing the resolution from $640 \times 480$ to $1000 \times 1000$ results in a speedup of 30%. Obviously, integrating support for $4 \times 4$ bundles combined with kd-tree entry search would yield a further significant performance boost.

High-quality rendering with interactive frame rates still requires the compute power of many PCs. Therefore, the distribution framework of Section 8.3 is used to combine the resources of a cluster of PCs. All measurements are carried out using a cluster of 24 dual AthlonMP 1800+ PCs with 512 MB memory each. All slaves are connected to a commodity 100 Mbit switch. The master uses a Gigabit uplink to the switch.

| | Office | Conference | Power Plant |
|---|---|---|---|
| $640 \times 480$ | 1.72 | 1.12 | 0.33 |
| $800 \times 600$ | 1.77 | 1.22 | 0.42 |
| $1000 \times 1000$ | 1.84 | 1.33 | 0.44 |
| $1600 \times 1200$ | 2.00 | 1.46 | 0.48 |

*Table 9.2: Performance in million rays per second on a AthlonMP 1800+ CPU at different resolutions with 16 VPLs, full shading and filtering.*

Shifting discontinuity filtering from the master to the slaves (see Section 9.5) allows for exclusively transferring pixel colors. This reduces the required network bandwidth per frame dramatically from $4 - 6$ MB to 912 KB. The combination of a largely reduced bandwidth and the efficient distribution framework (see Section 8.3) ensures high scalability and high frame rates. Figure 9.6 demonstrates the essentially linear scalability of up to 24 PCs (48 CPUs). This setup allows for reaching 20 fps at video resolution

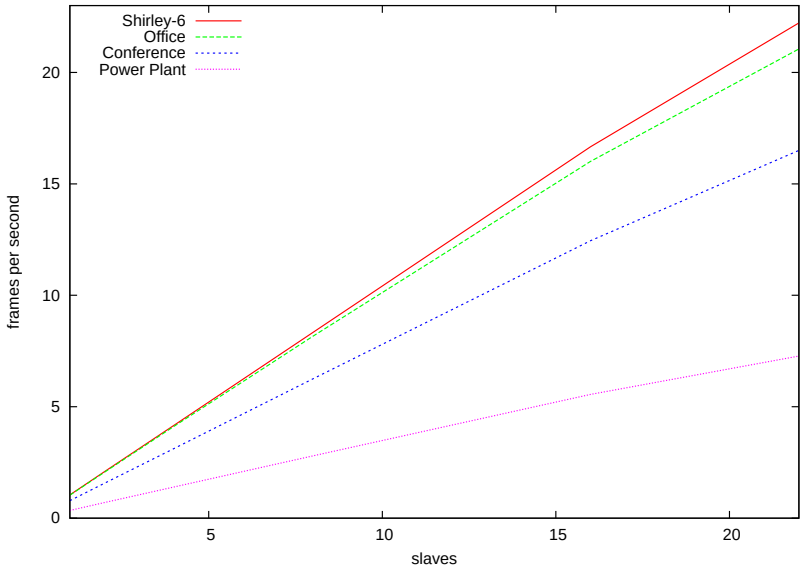



*Figure 9.6: System scalability: Performance is essentially linear for up to 24 PCs/48 CPUs. This applies to scenes ranging from several hundred triangles (Shirley-6) up to the Power Plant scene with 50 million triangles (four instances).*

($640 \times 480$) or 8 fps at full screen resolution ($1024 \times 1024$). Note that even for the power plant scene (four instances of 12.5 million triangles each), the complete illumination is recomputed from scratch for every single frame (see Figure 9.7).

## 9.7 Conclusions and Future Work

Besides adding support for missing illumination effects, e.g. caustics or glossy effects, it would be a great advance to optimize the system towards multi-core CPUs or shared memory setups. These two hardware setups are already becoming available on the desktop, and will provide enough compute power to achieve a performance level as the distributed setup. Through linear scalability in the number of processing resources and implicit support for coherent ray bundles, the instant global illumination system is the method of choice for realizing interactive global illumination.

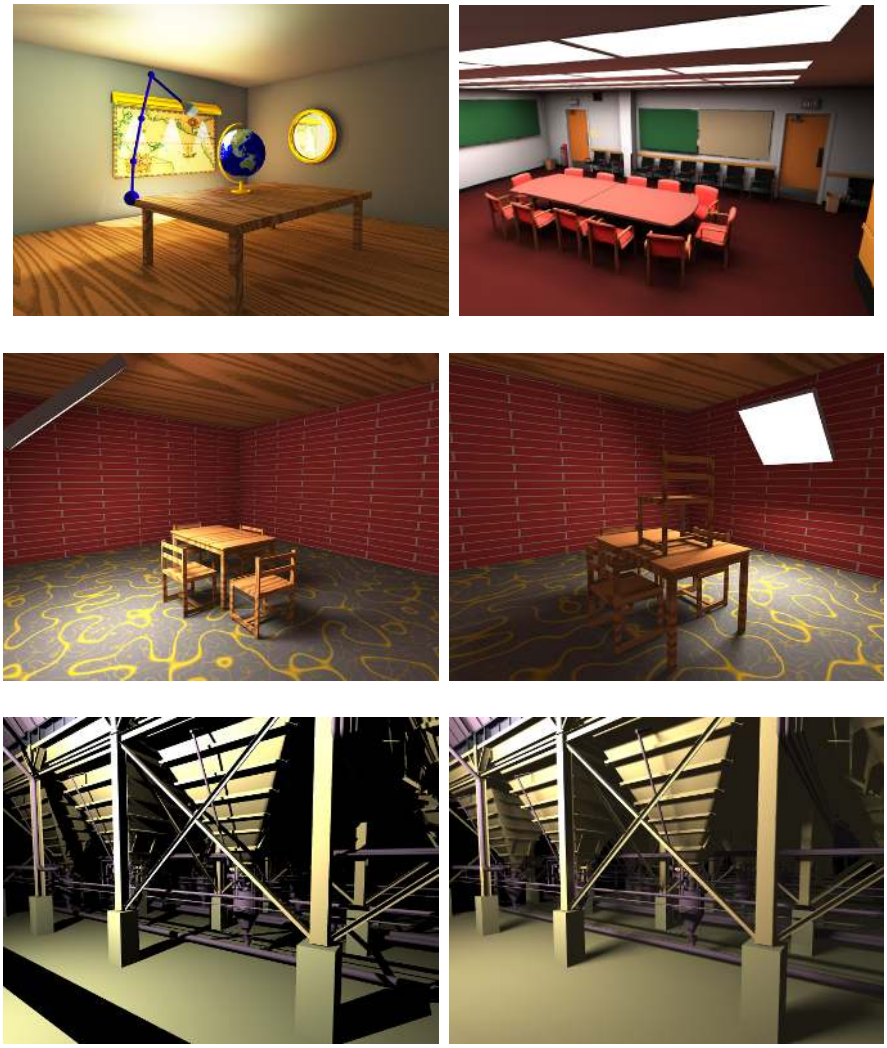


*Figure 9.7: Top left: Room with animated globe and lamp, containing roughly 20,000 triangles. The lamp illuminates North America which reflects a yellowish light into the room. Top right: The globally illuminated conference scene with* $280,000$ *triangles and* $104$ *light sources rendering at about* $20$ *fps on* $22$ *dual AthlonMP PCs with* $12$ *VPLs. Center left and right: The well-known Shirley-6 test scene consisting of* $600$ *triangles rendered with procedural shading. This scene is rendered at roughly* $22$ *fps with* $12$ *VPLs allowing arbitrary changes in the scene. Bottom left and right: A power plant scene containing four copies of the model with a total of* $50$ *million triangles rendered at about* $2$ *fps with global illumination. These two views clearly show the difference between direct illumination with hard ray traced shadows and the smooth lighting rendered by the global illumination system. All scenes were rendered at a resolution of* $640 \times 480$.

# Chapter 10

# Final Summary, Conclusions, and Future Work

In this final chapter, the content and all new contributions of this thesis will be briefly summarized. Before concluding, potential future work for all areas discussed here will be outlined.

In Chapter 2, the basics of ray tracing and the role of ray tracing as a rendering algorithm were discussed. After defining the core ray tracing algorithm which forms the basis for ray tracing-based rendering, the factors that impact its performance were presented. In particular, the need for an efficient traversal of rays through a spatial index structure and the need for an efficient primitive intersection test were discussed in detail. Based on this discussion, exploiting coherence, e.g. by traversing coherent ray bundles and an optimized implementation with respect to the underlying hardware architecture were identified as key factors to improve the performance of the core ray tracing algorithm.

As optimization in terms of the underlying hardware architecture plays a major role for achieving realtime ray tracing, a detailed discussion of the performance issues of current CPU architectures, e.g. cache misses, branch mis-predictions, and dependency chains, followed in Chapter 3. Based on this discussion, coding guidelines which help to achieve optimized code were proposed next. In addition, the SIMD extension of current CPUs was proposed as a powerful tool for exploiting the full compute power of current CPU architectures.

Chapter 4 used the optimization guidelines of Chapter 3 to describe several highly optimized example implementations for the traversal of coherent ray bundles through a kd-tree. In particular, implementations of different frustum traversal algorithms, which allow for constant traversal cost (in terms

of the number of rays within the frustum) were proposed.

The efficient ray bundle traversal algorithms presented in Chapter 4 could be directly used as the basis for ray tracing triangular scenes. Chapter 5 combined these traversal algorithms with fast triangle intersection algorithms. By relying on fast traversal and intersection algorithms designed for efficiently supporting ray bundles, even interactive ray tracing performance on a single CPU and at high resolutions was achieved.

Chapter 6 showed that interactive ray tracing performance is not limited to triangular scenes. In particular, highly optimized intersection algorithms, e.g. based on uniform refinement, Bézier clipping, and Newton iteration for efficiently supporting bicubic Bézier surfaces were presented. Each of these intersection algorithms has its advantages and disadvantages, so choosing a particular algorithm largely depends on performance and accuracy requirements.

Chapter 7 showed that the coding guidelines can be efficiently used to optimize the construction algorithm for high-quality kd-trees. The proposed algorithm, which even allows for multi-threaded kd-tree construction, is able to support ray tracing of fully dynamic scenes by reconstructing the corresponding kd-tree from scratch for every frame.

The need of ray tracing for compute power can be compensated by combining the compute power of multiple PCs. Chapter 8 presented a framework for parallel ray tracing using a cluster of off-the-shelf PCs. As the framework was exclusively designed for handling realtime ray tracing on a distributed memory environment, e.g. dealing with high interconnection latency and small interconnection bandwidth, linear scalability in the number of connected PCs was achieved.

Chapter 9 combined several techniques outlined in the preceding chapters, e.g. tracing of coherent ray bundles and the distribution framework, in order to realize a system that is even capable of interactively computing global illumination. The instant global illumination algorithm presented was adapted to directly benefit from coherent ray bundles, and extended using a high-performance shading framework. Moreover, the proposed implementation of the instant global illumination algorithm in terms of the distribution framework of Chapter 8 allowed for achieving linear performance scalability in the number of connected PCs.

## Future Work

Even though many aspects of potential future work have already been discussed throughout this thesis, they will be briefly summarized in the following.

Extremal traversal of coherent ray bundles allows for constant traversal cost, independently of the number of rays within the bundle. One should therefore concentrate on further reducing the intersection overhead caused by extremal traversal, as compared to that of a standard ray bundle traversal algorithm. A possible way to achieve this could be to automatically update the ray frustum during traversal by excluding already terminated rays.

For triangular scenes, it might be beneficial to store the geometry of triangles within a leaf as a small index face set. A potential ray bundle intersection would then first perform a side test based on the set of edges. Only for those triangles that fulfill the criteria of the side test, would a distance test be performed in a second step. This algorithm might be able to avoid a significant amount of unnecessary triangle intersection tests.

In terms of intersection tests for bicubic Bézier surfaces, future optimizations should concentrate on a Newton iteration-based intersection using the Krawczyk operator, because this is the only algorithm that reliably provides highly accurate results. As the cost and required iterations for the intersection algorithm largely depend on the accurate computation of the interval extension of the partial derivatives, further optimization should focus on this operation. It might also be beneficial to evaluate how the intersection algorithm can be efficiently extended to directly support coherent ray bundles.

Besides faster traversal and intersection, efficiently supporting dynamic scenes will be one of the major fields of future research for realtime ray tracing. The fast kd-tree construction algorithm proposed in this thesis can be seen as a first step; supporting scenes as used in today's computer games will require even further optimized algorithms.

Having a system capable of performing realtime ray tracing allows for directly exploring potential ways of realizing realtime global illumination. The modified instant global illumination algorithm presented in this thesis has been designed to optimally exploit coherent ray bundles. As the algorithm largely relies on casting shadow rays, future modifications could integrate extremal traversal for efficiently tracing a large number of shadow rays per bundle.

Besides the performance improvements made possible by using better algorithms, future hardware architectures will be able to offer an improved support for realtime ray tracing. Instead of increasing performance per CPU by raising the clock speed, future CPU architectures will improve performance by adding multiple cores to a single CPU chip. Even though current CPUs have two cores per CPU at the most, all major CPU manufacturers have already announced a higher number of cores for future CPU designs. A high number of cores per CPU is beneficial for ray tracing because the workload can be distributed across multiple cores. However, multi-core CPUs

present their own problems, e.g. limited interconnection bandwidth between cores, shared CPU caches, and efficient synchronization techniques. Future realtime ray tracing systems must consider these issues to exploit the full potential of multi-core CPUs.

A new design in terms of multi-core CPUs was introduced with the CELL processor [Cell05]. Instead relying on a homogeneous system of identical CPU cores, the CELL processor has a single standard CPU core and eight additional small CPU cores. These small CPUs have been exclusively designed for performing SIMD operations. Moreover, each of them has no memory cache but a small amount of local memory attached. If a small CPU core wants to access the main memory, a DMA request must be initiated manually. The new CELL architecture provides a significant amount of SIMD computing power, but requires new programming techniques, because all memory transfers must be initiated manually. However, a very early prototype implementation of a ray tracing system that relied on the algorithms proposed in this thesis was capable of ray tracing the conference scene (tracing only primary rays) with over 30 frames per second at a resolution of $1024 \times 1024$ on a 2.4 GHz CELL processor. Future optimization should concentrate on exploiting the full potential of the CELL chip, which might permit a ray tracing system to be created which offers similar compute power as an entire cluster of PCs.

## Final Conclusions

This thesis showed that algorithms designed and optimized in terms of the underlying processor architecture are the key factor for realizing realtime ray tracing. Even though for establishing realtime ray tracing as an alternative to rasterization-based rendering many open questions need to be answered realtime ray tracing is already starting to play a major role in 3D graphics.

Through the introduction of powerful multi-core processors, sufficient compute power will be available on everyone's desktop to explore the entire set of new applications, which now become possible with realtime ray tracing.

# Appendix A
# List of Related Papers

Many parts of this thesis have already been published in previous publications. This chapter provides a list of papers that have contributed to this thesis and that contain additional information.

### 2001

**Interactive Rendering using Coherent Ray Tracing**
*Ingo Wald, Carsten Benthin, Markus Wagner, and Philipp Slusallek*
*Computer Graphics Forum*, 20(3), 2001, pages 153–164, by A. Chalmers and T.–M. Rhyne (editors), Blackwell Publishers, Oxford, (Proceedings of Eurographics 2001), Manchester, UK [Wald01c]

**Interactive Distributed Ray Tracing of Highly Complex Models**
*Ingo Wald, Philipp Slusallek, and Carsten Benthin*
Rendering Techniques 2001 (Proceedings of the 12th Eurographics Workshop on Rendering), by Steven J. Gortler and Karol Myszkowski (editors), pages 274–285, pages 2001, London, UK [Wald01b]

### 2002

**Interactive Global Illumination using Fast Ray Tracing**
*Ingo Wald, Thomas Kollig, Carsten Benthin, Alexander Keller, Philipp Slusallek*, *Rendering Techniques 2002*, by P. Debevec and S. Gibson (editors) pages 15–24, 2002, Pisa, Italy, (Proceedings of the 13th Eurographics Workshop on Rendering) [Wald02b]

**Interactive Headlight Visualization –A Case Study of Interactive Distributed Ray Tracing–**
*Carsten Benthin, Ingo Wald, Tim Dahmen and Philipp Slusallek*, Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization (PVG), pages 81–88, Blaubeuren, Germany, 2002 [Benthin02]

**OpenRT – A Flexible and Scalable Rendering Engine for Interactive 3D Graphics**
*Ingo Wald, Carsten Benthin, and Philipp Slusallek*, Technical Report 2002, Saarland University Saarbrücken, Germany [Wald02a]

## 2003

**Towards Realtime Ray Tracing – Issues and Potential**
*Ingo Wald, Carsten Benthin, and Philipp Slusallek*, Technical Report 2003, Saarland University Saarbrücken, Germany [Wald03d]

**The OpenRT Application Programming Interface – Towards A Common API for Interactive Ray Tracing –**
*Andreas Dietrich, Ingo Wald, Carsten Benthin, and Philipp Slusallek*, OpenSG Symposium 2003, Darmstadt, Germany [Dietrich03]

**Interactive Global Illumination in Complex and Highly Occluded Scenes**
*Ingo Wald, Carsten Benthin, and Philipp Slusallek*, Proceedings of the 14th Eurographics Symposium on Rendering, by P. H. Christensen and D. Cohen-Or (editors) pages 74–81, 2003, Leuven, Belgium [Wald03c]

**Interactive Distributed Ray Tracing on Commodity PC Clusters – State of the Art and Practical Applications –**
*Ingo Wald, Carsten Benthin, and Philipp Slusallek*, in Harald Kosch, Laszlo Böszörmenyi, and Hermann Hellwagner, editors, *Euro-Par 2003*, Klagenfurt, Austria, volume 2790 of *Lecture Notes in Computer Science*, Springer. [Wald03a]

**A Scalable Approach to Interactive Global Illumination**
*Carsten Benthin, Ingo Wald, and Philipp Slusallek*, in *Computer Graphics Forum*, 22(3), 2003, pages, 621–630, (Proceedings of Eurographics 2003), Granada, Spain [Benthin03]

**Distributed Interactive Ray Tracing in Dynamic Environments**
*Ingo Wald, Carsten Benthin, and Philipp Slusallek*, SIGGRAPH/Eurographics Workshop on Parallel Graphics and Visualization (PVG) 2003, Seattle, WA, USA, pages 77-86 [Wald03b]

**Realtime Ray Tracing and its Use for Interactive Global Illumination**
*Ingo Wald, Timothy J.Purcell, Jörg Schmittler, Carsten Benthin, and Philipp Slusallek*, Eurographics 2003 State-of-the-Art Reports, Granada, Spain [Wald03e]

## 2004

**Interactive Ray Tracing of Freeform Surfaces**
*Carsten Benthin, Ingo Wald, and Philipp Slusallek*, Afrigraph 2004, Stellenbosch (Cape Town), South Africa [Benthin04]

## 2005

**A Ray Tracing based Framework for High-Quality Virtual Reality in Industrial Design Applications**
*Ingo Wald, Carsten Benthin, Alexander Efremov, Tim Dahmen, Johannes Guenther, Andreas Dietrich, Vlastimil Havran, Philipp Slusallek, Hans-Peter Seidel*, Technical Report 2005, University of Utah [Wald05]

**Techniques for Interactive Ray Tracing of Bézier Surfaces**
*Carsten Benthin, Ingo Wald, and Philipp Slusallek*, Journal of Graphics Tools (to appear) [Benthin05]

# Bibliography


[AltiVec] Motorola Inc. *AltiVec Technology Facts*. Available at http://www.motorola.com/AltiVec/facts.html.

[Amanatides87] *John Amanatides and Andrew Woo*. A Fast Voxel Traversal Algorithm for Ray Tracing. In *Proceedings of Eurographics*, pages 3–10. Eurographics Association, 1987.

[AMD03] *Advanced Micro Devices*. AMD Opteron Processor Model 8 Data Sheet. http://www.amd.com/us-en/Processors, 2003.

[Arvo87] *James Arvo and David Kirk*. Fast Ray Tracing by Ray Classification. *Computer Graphics (Proceedings of ACM SIGGRAPH)*, 21(4):55–64, 1987.

[Badouel92] *Didier Badouel*. An Efficient Ray Polygon Intersection. In David Kirk, editor, *Graphics Gems III*, pages 390–393. Academic Press, 1992.

[Benthin02] *Carsten Benthin, Ingo Wald, Tim Dahmen, and Philipp Slusallek*. Interactive Headlight Simulation – A Case Study of Distributed Interactive Ray Tracing. In *Proceedings of the 4th Eurographics Workshop on Parallel Graphics and Visualization (PGV)*, pages 81–88, 2002.

[Benthin03] *Carsten Benthin, Ingo Wald, and Philipp Slusallek*. A Scalable Approach to Interactive Global Illumination. *Computer Graphics Forum (Proceedings of Eurographics)*, 22(3):621–630, 2003.

[Benthin04] *Carsten Benthin, Ingo Wald, and Philipp Slusallek*. Interactive Ray Tracing of Free-Form Surfaces. In *Proceedings of Afrigraph*, pages 99–106, November 2004.

[Benthin05] *Carsten Benthin, Ingo Wald, and Philipp Slusallek.* Techniques for Interactive Ray Tracing of Bézier Surfaces. *Journal of Graphics Tools (to appear)*, 2005.

[Bittner99] *Jiri Bittner.* Hierarchical Techniques for Visibility Determination. Technical Report DS-005, Department of Computer Science and Engineering, Czech Technical University in Prague, 1999. Also available at http:/www.cgg.cvut.cz/∼bittner/publications/-minimum.ps.gz.

[Campagna97] *Swen Campagna, Philipp Slusallek, and Hans-Peter Seidel.* Ray Tracing of Parametric Surfaces. *The Visual Computer*, 13(6):265–282, 1997.

[Cell05] *International Business Machines.* The Cell Project at IBM Research. http://www.research.ibm.com/cell/, 2005.

[Chalmers98] *Alan Chalmers and Erik Reinhard.* Parallel and Distributed Photo-Realistic Rendering. In *Course notes for ACM SIGGRAPH*, pages 425–432. ACM Press, 1998.

[Chalmers01] *Alan Chalmers, Tim Davis, Toshi Kato, and Erik Reinhard.* Practical Parallel Processing for Today's Rendering Challenges. In *Course notes for ACM SIGGRAPH.* ACM Press, 2001.

[Cohen93] *Michael F. Cohen and John R. Wallace. Radiosity and Realistic Image Synthesis.* Morgan Kaufmann Publishers, 1993.

[Cohen94] *Daniel Cohen.* Voxel Traversal along a 3D Line. In Paul Heckbert, editor, *Graphics Gems IV*, pages 366–369. Academic Press, 1994.

[Cook84a] *Robert Cook, Thomas Porter, and Loren Carpenter.* Distributed Ray Tracing. *Computer Graphics (Proceeding of ACM SIGGRAPH)*, 18(3):137–144, 1984.

[Cook84b] *Robert L. Cook.* Shade trees. *Computer Graphics (Proceedings of ACM SIGGRAPH)*, 18(3):223–231, 1984.

[DeMarle03] *David E. DeMarle, Steve Parker, Mark Hartner, Christiaan Gribble, and Charles Hansen.* Distributed Interactive Ray Tracing for Large Volume Visualization. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG)*, pages 87–94, 2003.

[DeMarle04] *David E. DeMarle, Christiaan Gribble, and Steven Parker.* Memory-Savvy Distributed Interactive Ray Tracing. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 93–100, 2004.

[Dietrich03] *Andreas Dietrich, Ingo Wald, Carsten Benthin, and Philipp Slusallek.* The OpenRT Application Programming Interface – Towards A Common API for Interactive Ray Tracing. In *Proceedings of the 2003 OpenSG Symposium*, pages 23–31, 2003.

[Dietrich05] *Andreas Dietrich, Carsten Colditz, Oliver Deussen, and Philipp Slusallek.* Realistic and Interactive Visualization of High-Density Plant Ecosystems. In *Natural Phenomena 2005, Proceedings of the Eurographics Workshop on Natural Phenomena*, pages 73–81, August 2005.

[Dutre03] *Phil Dutre, Philippe Bekaert, and Kavita Bala. Advanced Global Illumination.* A K Peters, 1st edition, July 2003.

[Efremov05] *Alexander Efremov, Vlastimil Havran, and Hans-Peter Seidel.* Robust and Numerically Stable Bézier Clipping Method for Ray Tracing NURBS Surfaces. In *SCCG'05 Proceedings*, 2005.

[Erickson97] *Jeff Erickson.* Pluecker Coordinates. *Ray Tracing News*, 1997. http://www.acm.org/tog/resources/-RTNews/html/rtnv10n3.html#art11.

[Farin96] *G. Farin. Curves and Surfaces for Computer Aided Geometric Design 4th Edition.* Academic Press, Boston, 1996.

[Foley97] *Foley, van Dam, Feiner, and Hughes. Computer Graphics – Principles and Practice, 2nd edition.* Addison Wesley, 1997.

[Foley05] *Tim Foley and Jeremy Sugerman.* KD-tree Acceleration Structures for a GPU Raytracer. In *HWWS '05 Proceedings*, pages 15–22. ACM Press, 2005.

[Foruma] *MPI Forum.* MPI – The Message Passing Interface Standard. http://www-unix.mcs.anl.gov/mpi.

[Forumb] *Myrinet Forum.* Myrinet. http://www.myri.com/-myrinet/overview/.

[Fujimoto86] *Akira Fujimoto, Takayuki Tanaka, and Kansei Iwata.* ARTS: Accelerated Ray Tracing System. *IEEE Computer Graphics and Applications*, 6(4):16–26, 1986.

[Futral01] *William T. Futral. Infiniband Architecture: Development and Deployment – A Strategic Guide to Server I/O Solutions.* Intel Press, 2001.

[Geimer05] *Markus Geimer and Oliver Abert.* Interactive Ray Tracing of Trimmed Bicubic Bézier Surfaces without Triangulation. In *WSCG (Full Papers)*, pages 71–78, 2005.

[Geist94] *Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidyalingam S. Sunderam. PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Network Parallel Computing.* MIT Press, Cambridge, 1994.

[Gigante88] *Michael Gigante.* Accelerated Ray Tracing using Non-Uniform Grids. In *Proceedings of Ausgraph*, pages 157–163, 1988.

[Glassner84] *Andrew S. Glassner.* Space Subdivision For Fast Ray Tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, 1984.

[Glassner89] *Andrew Glassner. An Introduction to Ray Tracing.* Morgan Kaufmann, 1989.

[Glassner95] *Andrew Glassner. Principles of Digital Image Synthesis.* Morgan Kaufmann, 1995.

[GNU] The GNU gcc compiler, version 3.4.X. http://www.gnu.org.

[Goldsmith87] *Jeffrey Goldsmith and John Salmon.* Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics and Applications*, 7(5):14–20, May 1987.

[Guthe02] *M. Guthe, J. Meseth, and R. Klein.* Fast and Memory Efficient View-Dependent Trimmed NURBS Rendering. *Pacific Graphics 2002*, pages 204–213, October 2002.

[Haines91] *Eric Haines.* Efficiency Improvements for Hierarchy Traversal in Ray Tracing. In James Arvo, editor, *Graphics Gems II*, pages 267–272. Academic Press, 1991.

[Havran01] *Vlastimil Havran.* Heuristic Ray Shooting Algorithms. PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, 2001.

[Hsiung92] *Ping-Kang Hsiung and Robert H. Thibadeau.* Accelerating ARTS. *The Visual Computer*, 8(3):181–190, March 1992.

[IBM05] *International Business Machines.* IBM Power5. http://www03.ibm.com/systems/power/, 2005.

[Intel97] *Intel Corp.* Using the RDTSC Instruction for Performance Monitoring. http://www.intel.com/drg/-pentiumII/appnotes/RDTSCPM1.HTM, 1997.

[Intel01] Intel Corp. *IA-32 Intel Architecture Optimization – Reference Manual*, 2001.

[Intel02a] Intel Corp. *Intel C/C++ Compilers*, 2002. http://www.intel.com/software/products/compilers.

[Intel02b] *Intel Corp.* Intel Pentium III Streaming SIMD Extensions. http://developer.intel.com/vtune/cbts/simd.htm, 2002.

[Intel02c] *Intel Corp.* Introduction to Hyper-Threading Technology. http://developer.intel.com/technology/hyperthread, 2002.

[Intel03] Intel Corp. *Prescott New Instructions*, 2003. http://www.intel.com/cd/ids/developer/asmo-na/eng/microprocessors/ia32/pentium4/resources/.

[Intel04] *Intel Corp.* Intel VTune Performance Analyzers. http://developer.intel.com/software/products/-vtune/index.htm, 2004.

[Intel05] *Intel Corp.* Intel Next Generation Micro Architecture. http://www.intel.com/technology/computing/ngma/, 2005.

[Jensen01] *Henrik Wann Jensen. Realistic Image Synthesis Using Photon Mapping.* A K Peters, 2001.

[Jevans89] *David Jevans and Brian Wyvill.* Adaptive Voxel Subdivision for Ray Tracing. *Proceedings of Graphics Interface '89*, pages 164–172, 1989.

[Kay86] *Timothy L. Kay and James T. Kajiya.* Ray Tracing Complex Scenes. *Computer Graphics (Proceedings of ACM SIGGRAPH)*, 20(4):269–278, 1986.

[Keates95] *Martin J. Keates and Roger J. Hubbold.* Interactive Ray Tracing on a Virtual Shared-Memory Parallel Computer. *Computer Graphics Forum*, 14(4):189–202, 1995.

[Keller97] *Alexander Keller.* Instant Radiosity. *Computer Graphics (Proceedings of ACM SIGGRAPH)*, pages 49–56, 1997.

[Keller01] *Alexander Keller and Wolfgang Heidrich.* Interleaved Sampling. *Rendering Techniques*, pages 269–276, 2001. (Proceedings of the 12th Eurographics Workshop on Rendering).

[Kirk91] *David Kirk and James Arvo.* Improved Ray Tagging For Voxel-Based Ray Tracing. In James Arvo, editor, *Graphics Gems II*, pages 264–266. Academic Press, 1991.

[Klimaszewski97] *Kryzsztof S. Klimaszewski and Thomas W. Sederberg.* Faster Ray Tracing using Adaptive Grids. *IEEE Computer Graphics and Applications*, 17(1):42–51, January/ February 1997.

[Krawczyk69] *R. Krawczyk.* Newton-Algorithmen zur Bestimmung von Nullstellen mit Fehlerschranken. *Computing*, 4:187–201, 1969.

[Krawczyk70] *R. Krawczyk.* Einschiebung von Nullstellen mit Hilfe einer Intervallarithmetik. *Computing*, 5:356–370, 1970.

[Lafortune93] *Eric Lafortune and Yves Willems.* Bidirectional Path Tracing. In *Proc. 3rd International Conference on Computational Graphics and Visualization Techniques (Compugraphics)*, pages 145–153, 1993.

[Lewis02] *R. Lewis, R. Wang, and D. Hung.* Design of a Pipelined Architecture for Ray/Bezier Patch Intersection Computation. *Canadian Journal of Electrical and Computer Engineering*, 28(1), 2002.

[Lext01] *Jonas Lext and Tomas Akenine-Möller.* Towards Rapid Reconstruction for Animated Ray Tracing. In *Eurographics 2001 – Short Presentations*, pages 311–318, 2001.

[MacDonald89] *J. David MacDonald and Kellogg S. Booth.* Heuristics for Ray Tracing using Space Subdivision. In *Proceedings of Graphics Interface*, pages 152–63, 1989.

[MacDonald90] *J. David MacDonald and Kellogg S. Booth.* Heuristics for Ray Tracing using Space Subdivision. *Visual Computer*, 6(6):153–65, 1990.

[Martin00] *W. Martin, E. Cohen, R. Fish, and P. Shirley.* Practical Ray Tracing of Trimmed NURBS Surfaces. *Journal of Graphics Tools*, 5:27–52, 2000.

[Möller97] *Tomas Möller and Ben Trumbore.* Fast, Minimum Storage Ray Triangle Intersection. *Journal of Graphics Tools*, 2(1):21–28, 1997.

[Moore77] *R. E. Moore and S.T. Jones.* Save Starting Regions for Iterative Methods. *SIAM J. Numer. Anal.*, 14:1051–1065, 1977.

[Muuss95a] *Michael J. Muuss.* Towards Real-Time Ray-Tracing of Combinatorial Solid Geometric Models. In *Proceedings of BRL-CAD Symposium*, 1995.

[Muuss95b] *Michael J. Muuss and Maximo Lorenzo.* High-Resolution Interactive Multispectral Missile Sensor Simulation for ATR and DIS. In *Proceedings of BRL-CAD Symposium*, 1995.

[Nichols96] *Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell.* *PThreads Programming.* O'Reilly, 1996.

[Nishita90] *T. Nishita, T. W. Sederberg, and M. Kakimoto.* Ray Tracing Trimmed Rational Surface Patches. *Computer Graphics (Proceedings of ACM SIGGRAPH)*, pages 337–345, 1990.

[Parker98] *Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter-Pike Sloan.* Interactive Ray Tracing for Isosurface Rendering. In *IEEE Visualization*, pages 233–238, October 1998.

[Parker99a] *Steven Parker, Michael Parker, Yarden Livnat, Peter-Pike Sloan, Chuck Hansen, and Peter Shirley.* Interactive Ray Tracing for Volume Visualization. *IEEE Transactions on Computer Graphics and Visualization*, 5(3):238–250, 1999.

[Parker99b] *Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter-Pike Sloan.* Interactive Ray Tracing. In *Proceedings of Interactive 3D Graphics*, pages 119–126, 1999.

[Pharr04] *Matt Pharr and Greg Humphreys. Physically Based Rendering : From Theory to Implementation.* Morgan Kaufman, 2004.

[Piegl97] *Les Piegl and Wayne Tiller. The NURBS book, 2nd edition.* Springer-Verlag, Inc., 1997.

[Purcell02] *Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan.* Ray Tracing on Programmable Graphics Hardware. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)*, 21(3):703–712, 2002.

[Ramsey04] *Shaun Ramsey, Kristin Potter, and Charles Hansen.* Ray Bilinear Patch Intersections. *Journal of Graphics Tools*, 9(3):41–47, 2004.

[Reinhard95] *Erik Reinhard.* Scheduling and Data Management for Parallel Ray Tracing. PhD thesis, University of East Anglia, 1995.

[Reinhard97] *Erik Reinhard and Frederik W. Jansen.* Rendering Large Scenes using Parallel Ray Tracing. *Parallel Computing*, 23(7):873–885, 1997.

[Reshetov05] *Alexander Reshetov, Alexei Soupikov, and Jim Hurley.* Multi-Level Ray Tracing Algorithm. *ACM Trans. Graph.*, 24(3):1176–1185, 2005.

[Rubin80] *Steve M. Rubin and Turner Whitted.* A Three-Dimensional Representation for Fast Rendering of Complex Scenes. *Computer Graphics*, 14(3):110–116, July 1980.

[Samet89] *Hanan Samet.* Implementing Ray Tracing with Octrees and Neighbor Finding. *Computers and Graphics*, 13(4):445–60, 1989.

[Schmittler02] *Jörg Schmittler, Ingo Wald, and Philipp Slusallek.* SaarCOR – A Hardware Architecture for Ray Tracing. In *Proceedings of the ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, pages 27–36, 2002.

[Schmittler04] *Jörg Schmittler, Sven Woop, Daniel Wagner, Wolfgang J. Paul, and Philipp Slusallek.* Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip. In *Proceedings of Graphics Hardware*, 2004.

[Shirley02] *Peter Shirley. Fundamentals of Computer Graphics.* A K Peters, 2002.

[Shirley03] *Peter Shirley and R. Keith Morley. Realistic Ray Tracing.* A K Peters, Second edition, 2003.

[Shoemake98] *Ken Shoemake.* Pluecker Coordinate Tutorial. *Ray Tracing News*, 1998. http://www.acm.org/tog/resources/-RTNews/html/rtnv11n1.html#art3.

[Simiakakis95] *George Simiakakis.* Accelerating Ray Tracing with Directional Subdivision and Parallel Processing. PhD thesis, University of East Anglia, 1995.

[Smits98] *Brian Smits.* Efficiency Issues for Ray Tracing. *Journal of Graphics Tools*, 3(2):1–14, 1998.

[Stevens98] *W. Richard Stevens.* *Unix Network Programming Volume 1.* Prentice Hall, 1998.

[Subramanian90a] *K. R. Subramanian.* A Search Structure based on kd-Trees for Efficient Ray Tracing. PhD thesis, University of Texas at Austin, 1990.

[Subramanian90b] *K. R. Subramanian and Donald S. Fussel.* Factors Affecting Performance of Ray Tracing Hierarchies. Technical report, The University of Texas at Austin, 1990.

[Sung92] *Kelvin Sung and Peter Shirley.* Ray Tracing with the BSP Tree. In David Kirk, editor, *Graphics Gems III*, pages 271—274. Academic Press, 1992.

[Sweeney86] *M. Sweeney and R. Bartels.* Ray Tracing Free-Form B-Spline Surfaces. *IEEE Computer Graphics and Applications*, 6(3):41–49, 1986.

[Toth85] *Daniel L. Toth.* On Ray Tracing Parametric Surfaces. In *Proceedings of ACM Siggraph*, pages 171–179, New York, NY, USA, 1985. ACM Press.

[Veach94] *Eric Veach and Leonid Guibas.* Bidirectional Estimators for Light Transport. In *Proceedings of the 5th Eurographics Workshop on Rendering*, pages 147 – 161, 1994.

[Veach97] *Eric Veach and Leonid Guibas.* Metropolis Light Transport. In Turner Whitted, editor, *Proceedings of ACM SIGGRAPH*, pages 65–76, 1997.

[Wald01a] *Ingo Wald, Philipp Slusallek, and Carsten Benthin.* Interactive Distributed Ray Tracing of Highly Complex Models. In *Rendering Techniques*, pages 274–285, 2001. (Proceedings of Eurographics Workshop on Rendering).

[Wald01b] *Ingo Wald, Philipp Slusallek, and Carsten Benthin.* Interactive Distributed Ray Tracing of Highly Complex Models. In Steven J. Gortler and Karol Myszkowski, editors, *Rendering Techniques*, Proceedings of the 12th Eurographics Workshop on Rendering Techniques, London, UK, June 25-27, 2001, pages 274–285. Springer, 2001.

[Wald01c] *Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner.* Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum*, 20(3):153–164, 2001. (Proceedings of Eurographics).

[Wald02a] *Ingo Wald, Carsten Benthin, and Philipp Slusallek.* OpenRT - A Flexible and Scalable Rendering Engine for Interactive 3D Graphics. Technical report, Saarland University, 2002. Available at http://graphics.cs.uni-sb.de/Publications.

[Wald02b] *Ingo Wald, Thomas Kollig, Carsten Benthin, Alexander Keller, and Philipp Slusal lek.* Interactive Global Illumination using Fast Ray Tracing. In Paul Debevec and Simon Gibson, editors, *Rendering Techniques 2002*, pages 15–24, Pisa, Italy, June 2002. Eurographics Association, Eurographics. (Proceedings of the 13th Eurographics Workshop on Rendering).

[Wald03a] *Ingo Wald, Carsten Benthin, Andreas Dietrich, and Philipp Slusallek.* Interactive Ray Tracing on Commodity PC Clusters – State of the Art and Practical Applications. In Harald Kosch, László Böszörményi, and Hermann Hellwagner, editors, *Euro-Par*, volume 2790 of *Lecture Notes in Computer Science*, pages 499–508. Springer, 2003.

[Wald03b] *Ingo Wald, Carsten Benthin, and Philipp Slusallek.* Distributed Interactive Ray Tracing of Dynamic Scenes. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG)*, 2003.

[Wald03c] *Ingo Wald, Carsten Benthin, and Philipp Slusallek.* Interactive Global Illumination in Complex and Highly Occluded Environments. In Per H Christensen and Daniel Cohen-Or, editors, *Proceedings of the 2003 Eurographics Symposium on Rendering*, pages 74–81, Leuven, Belgium, 2003.

[Wald03d] *Ingo Wald, Carsten Benthin, and Philipp Slusallek.* Towards Realtime Ray Tracing – Issues and Potential. Technical report, Saarland University, 2003.

[Wald03e] *Ingo Wald, Timothy J. Purcell, Jörg Schmittler, Carsten Benthin, and Philipp Slusallek.* Realtime Ray Tracing and its Use for Interactive Global Illumination. In *Eurographics State of the Art Reports*, pages 85–122, 2003.

[Wald04] *Ingo Wald.* Realtime Ray Tracing and Interactive Global Illumination. PhD thesis, Computer Graphics Group, Saarland University, 2004.

[Wald05] *Ingo Wald, Carsten Benthin, Alexander Efremov, Tim Dahmen, Johannes Guenther, Andreas Dietrich, Vlastimil Havran, Philipp Slusallek, and Hans-Peter Seidel.* A Ray Tracing based Framework for High-Quality Virtual Reality in Industrial Design Applications. Technical Report UUSCI-2005-009, SCI Institute, University of Utah, 2005.

[Wang01] *S. Wang, Z. Shih, and R. Chang.* An Efficient and Stable Ray Tracing Algorithm for Parametric Surfaces. *18th Journal of Information Science and Engineering*, pages 541–561, 2001.

[Whang95] *K. Y. Whang, J. W. Song, J. W. Chang, J. Y. Kim, W. S. Cho, C. M. Park, and I. Y. Song.* Octree-R: An Adaptive Octree for efficient Ray Tracing. *IEEE Transactions on Visualization and Computer Graphics*, 1(4):343–349, 1995.

[Whitted80] *Turner Whitted.* An Improved Illumination Model for Shaded Display. *CACM*, 23(6):343–349, 1980.

[Woodward89] *C. Woodward.* Ray Tracing of Parametric Surfaces by Subdivision in the Viewing Plane. *Theory and Practice of Geometric Modeling*, 1989.

[Woop05] *Sven Woop, Joerg Schmittler, and Philipp Slusallek.* RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. *Proceedings of ACM SIGGRAPH*, 2005.