Ray Tracing Techniques for Computer Games and Isosurface Visualization

Heiko Friedrich

Computer Graphics Group
Saarland University
Saarbrücken, Germany

Thesis for obtaining the title of
Doctor of Engineering
of the Faculties of Natural Sciences and
Technology of Saarland University

Betreuender Hochschullehrer / Supervisors:
Prof. Dr. Philipp Slusallek, Universität des Saarlandes,
Saarbrücken, Germany

Gutachter / Reviewers:
Prof. Dr. Philipp Slusallek, Universität des Saarlandes,
Saarbrücken, Germany
Dr. Karol Myszkowski, Max-Planck-Institut für Informatik,
Saarbrücken, Germany

Dean:
Prof. Dr. Mark Groves, Universität des Saarlandes,
Saarbrücken, Germany

Eingereicht am / Thesis submitted:
3. Juni 2009 / June 3rd 2009

Datum des Kolloquium / Date of defense:
21. Oktober 2011 / October 21st 2011

Vorsitzender des Kolloquiums:
Prof. Dr. Joachim Weickert

Wissenschaftlicher Beisitzer:
Dr. Vincent Pegoraro

Heiko Friedrich
Universität des Saarlandes
Fachrichtung 6.2 - Informatik
Im Stadtwald - Building E 1 1, Room 013
66123 Saarbrücken

Abstract

Ray tracing is a powerful image synthesis technique, that has been used for high-quality offline rendering since decades. In recent years, this technique has become more important for realtime applications, but still plays only a minor role in many areas. Some of the reasons are that ray tracing is compute intensive and has to rely on preprocessed data structures to achieve fast performance. This dissertation investigates methods to broaden the applicability of ray tracing and is divided into two parts.

The first part explores the opportunities offered by ray tracing based game technology in the context of current and expected future performance levels. In this regard, novel methods are developed to efficiently support certain kinds of dynamic scenes, while avoiding the burden to fully recompute the required data structures. Furthermore, todays ray tracing performance levels are below what is needed for 3D games. Therefore, the multi-core CPU of the Playstation 3 is investigated, and an optimized ray tracing architecture presented to take steps towards the required performance.

In part two, the focus shifts to isosurface raytracing. Isosurfaces are particularly important to understand the distribution of certain values in volumetric data. Since the structure of volumetric data sets is diverse, optimized algorithms and data structures are developed for rectilinear as well as unstructured data sets which allow for realtime rendering of isosurfaces including advanced shading and visualization effects. This also includes techniques for out-of-core and time-varying data sets.

Kurzfassung

Ray-tracing ist ein flexibles Bildgebungsverfahren, das schon seit Jahrzehnten für hoch qualitative, aber langsame Bilderzeugung genutzt wird. In den letzten Jahren wurde Ray-tracing auch für Echtzeitanwendungen immer interessanter, spielt aber in vielen Anwendungsbereichen noch immer eine untergeordnete Rolle. Einige der Gründe sind die Rechenintensität von Ray-tracing sowie die Abhängigkeit von vorberechneten Datenstrukturen um hohe Geschwindigkeiten zu erreichen. Diese Dissertation untersucht Methoden um die Anwendbarkeit von Ray-tracing in zwei verschiedenen Bereichen zu erhöhen.

Im ersten Teil dieser Dissertation werden die Möglichkeiten, die Ray-tracing basierte Spieletechnologie bietet, im Kontext mit aktueller sowie zukünftig erwarteten Geschwindigkeiten untersucht. Darüber hinaus werden in diesem Zusammenhang Methoden entwickelt um bestimmte zeitveränderliche Szenen darstellen zu können ohne die dafür benötigen Datenstrukturen von Grund auf neu erstellen zu müssen. Da die Geschwindigkeit von Ray-tracing für Spiele bisher nicht ausreichend ist, wird die Mehrkern-CPU der Playstation 3 untersucht, und ein optimiertes Ray-tracing System beschrieben, das Ray-tracing näher an die benötigte Geschwindigkeit heranbringt.

Der zweite Teil beschfäftigt sich mit der Darstellung von Isoflächen mittels Ray-tracing. Isoflächen sind insbesonders wichtig um die Verteilung einzelner Werte in volumetrischen Datensätzen zu verstehen. Da diese Datensätze verschieden strukturiert sein können, werden für gitterförmige und unstrukturierte Datensätze optimierte Algorithmen und Datenstrukturen entwickelt, die die Echtzeitdarstellung von Isoflächen erlauben. Dies beinhaltet auch Erweiterungen für extrem große und zeitveränderliche Datensätze.

Acknowledgments

I would like to thank a number of people for their help on this dissertation. Working on optimized large scale software systems is always teamwork. First of all, I would like to thank my supervisor Prof. Dr. Philipp Slusallek. He guided me during my PhD, pushed me forward, inspired me, and helped me with discussions and ideas. Another thanks I owe my (former) colleagues, (in alphabetical order): Tim Dahmen, Georg Demme, Andreas Dietrich, Johannes Günther, Iliyan Georgiev, Krzysztof Kobus, Dr. Marco Lohse, Dr. Gerd Marmitt, Dr. Andreas Pohmi, Stefan Popov, Michael Repplinger, Dmitri Rubinstein, Michael Scherbaum, Dr. Jörg Schmittler, Dr. Sven Woop, and Hanna Schilt, the secretary of our computer graphics group.

In particular I have to thank Dr. Ingo Wald and Dr. Carsten Benthin. They helped me a lot with ideas, discussion and programming on all my projects. Without their encouraging support and a countless number of discussions through the last years I would not have been able to complete this dissertation.

Furthermore I want to thank my colleagues and friends from the Max-Plank-Institute for Computer Science (MPII) at the department AG4. Special thanks also goes to Karol Myszkowski at the MPII for reviewing this dissertation, and the post graduate programme *Leistungsgarantien für Rechnersysteme* as well as the Deutsche Forschungsgemeinschaft for supporting my PhD studies with a scholarship. Another thanks goes to the SysAdmin Team of the computer graphics group. Finally, and most importantly, I want to thank my parents and friends who supported me all time.

We cannot change the cards that we are dealt,
just how we play the hand.

Randy Pausch

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Ray tracing is a powerful image synthesis technique which has been used for high-quality offline rendering since decades. An outstanding feature of ray tracing is its algorithmic simplicity and elegance. In a well-designed ray tracing system it is possible to integrate different kinds of graphics primitives and shading models such that they can smoothly interact with each other. Furthermore, the underlying ray concept allows for arbitrary recursive visibility queries which can be used for e.g. shadow, reflection, refraction, or even global illumination computations. However, ray tracing is compute intensive and has to rely on precomputed spatial-index structures for fast performance which limits its applicability in certain areas.

In recent years, researchers were able to lift ray tracing performance to interactive levels e.g. first by the use of massive parallel supercomputers [KH95, Muu95, PMS+99], later on commodity PCs by exploiting more suitable and optimized acceleration structures [GS87, MB90, Hav01a, Wal04], and highly optimized implementations and algorithmic improvements [Wal04, RSH05, Ben06].

But still, on current desktop machines the performance is below what is needed for certain application. For example, 3D computer games require typically frame rates of more than 30Hz. On todays mainstream CPUs and GPUs this is not realistic for complex scenes and shading. In other domains like scientific visualization it is similar.

Even worse, due to the fact that ray tracing relies on acceleration structures for fast performance, it is in general not possible to change a scenes geometry without invalidating the corresponding acceleration structure. When primitives are transformed it is most likely that the acceleration structure that is build over the initial geometry is not valid for the changed geometry. Current interactive ray tracing implementations are limited to hierarchical affine transformations of rigid bodies [WBS03]. For more complex animations, the corresponding acceleration structures would have to be rebuild from scratch. Unfortunately, acceleration structures cannot be build as fast as it would be required for 3D computer games or interactive scientific visualization applications. Even with the fastest known algorithms [Ben06], acceleration structures can only be rebuild a few times per second for moderately complex scenes.

Additionally, the size of the acceleration structures itself can be problematic. For example, volumetric datasets tend to be large by itself and the required acceleration structures can easily exceed the available main memory of a system.

Finally realtime ray tracing implementations require optimized code that exploits the features of a specific CPU or GPU for best performance.

1.1 The Goals of this Dissertation

This dissertation concerns itself with several of the above mentioned problems in different application areas of ray tracing. In short, the goal is to present solutions that allow for a use of ray tracing in 3D computer games and isosurface visualization applications.

More specifically, some of the above mentioned problems will be addressed by the following contributions and are the result of joint work with several other researchers (see Appendix A for more details):

Ray Tracing for Future 3D Computer Games: Explores the opportunities offered by ray tracing based game technology in the context of current and expected future performance levels. In particular, simulation based graphics that avoids pre-computations and thus enables the interactive production of advanced visual effects is discussed, the advantages of ray tracing for future computer games analyzed, ray tracing API issues illustrated, and first results from several ray tracing based game projects presented.

Ray Tracing of Dynamic Scenes: 3D computer games cannot life without dynamic content. Most importantly, support for animated characters is necessary. This thesis presents solutions to efficiently ray trace keyframe as well as skinned animations. The algorithm works by introducing a preprocess that identifies regions of locally coherent motion in a scene. For each of these regions, the motion can then be divided into an affine transformation and some residual motion that is captured by fuzzy boxes. Using this approach, a classic two-level hierarchy can be used to ray trace complex animated objects.

Ray Tracing on IBMs Cell: Current performance levels are below what is required for ray traced games. Therefore, the multicore-CPU of the Playstation 3 is investigated and an optimized ray tracing system described that makes full use of the processors features. Additionally, efficient software multithreading and caching is exploited to hide latencies for memory fetches.

Both, multithreading and caching are not hardware supported on a Cell's SPEs and require a careful software implementation in order to achieve fast ray shooting performance.

Isosurface Ray Tracing of Rectilinear Datasets: In scientific visualization, isosurfaces are of particular interest. In order to efficiently ray trace isosurfaces of rectilinear datasets the implicit min/max kD-tree as well as a fast analytic ray isosurface intersection test are presented. The acceleration structure does not require much memory and is also extended for the use of out-of-core datasets.

Isosurface Ray Tracing of Unstructured Datasets: For isosurfaces of unstructured datasets the implicit min/max BVH is introduced as well as a fast intersection test that relies on features of the marching tetrahedron algorithm. A simple update mechanism is also described which allows to efficiently render time-varying datasets.

1.2 The Structure of this Document

This dissertation consists of two parts. Part one deals with ray tracing for future computer games in the chapters two to five. Part two concerns itself with isosurface ray tracing in the chapters six to eight.

Chapter 2 gives a brief introduction to ray tracing and describes some of the fundamental differences between rasterization and ray tracing. After a description of the most important terms, the evolution of ray tracing is sketched. Finally, fast ray traversal and ray triangle intersection tests are outlined.

Chapter 3 explores the opportunities offered by ray tracing based game technology in the context of current and expected future performance levels. It describes the benefits of ray tracing for future computer games as well as prototype game implementations using ray tracing, and identifies problems that have to be solved for a serious use of ray tracing in computer games.

Chapter 4 describes a novel preprocess to efficiently ray trace dynamic scenes that are either defined by a set of keyframes, or skinned animations. The preprocess exploits the property of locally coherent motion that can be found in most 3D computer game like animations.

Chapter 5 analyzes the properties of the current Playstation 3 CPU and presents an optimized ray tracing algorithm. The implementation features software multithreading as well as manual data caching to hide the long memory fetch latencies.

Chapter 6 introduces the basic terminology for volume rendering, volumetric datasets, and in particular isosurface ray tracing. Furthermore, the evolution of isosurface rendering is briefly sketched and fundamental techniques for isosurface rendering are discussed.

Chapter 7 presents a compact kD-tree based acceleration structure to ray trace isosurfaces of rectilinear volumetric data sets which only require a low memory footprint. Additionally, a view independent extension for out-of-core isosurface ray tracing is proposed, and a novel fast and exact analytic ray isosurface intersection test is presented.

Chapter 8 describes an optimized system to ray trace isosurfaces in unstructured volumetric datasets. The system uses a tetrahedrization of the input data to build an implicit BVH and performs ray isosurface intersection tests by exploiting marching tetrahedron techniques. A benefit of using a BVH for unstructured volumetric data sets is that this allows to easily support time-varying datasets.

Chapter 9 summarizes again the contributions and outlines some directions for future work before final conclusions are drawn.

Part I

Ray Tracing for Computer Games

Chapter 2

An Introduction to Ray Tracing

This chapter begins with a quick comparison of the rasterization and ray tracing algorithm. After describing the core concept of ray tracing and how it can be used for image synthesis, the evolution of ray tracing as a high quality rendering approach is sketched. Finally, the most important concepts to achieve realtime ray tracing performance are laid out. As this is just a brief introduction, only the core ideas are described. Further information can be found e.g. in books [FvDFH97, Gla89, SM03, PH04] as well as PhD theses [Hav01a, Wal04, Ben06].

2.1 Rasterization and Ray Tracing

There are two major algorithms in computer graphics to synthesize 2D images from 3D *surface* descriptions e.g. defined by triangles. On the one hand is the *rasterization* approach, which is implemented in all off-the-shelf graphics cards. Due to the simplicity of the rasterization algorithm its core operations can be efficiently mapped to special purpose hardware and achieve extremely high rendering performance.

Rasterization operates sequentially on the geometric primitives of a scene. First, a geometric primitive is projected from its world-space coordinates onto the image plane. Then, the set of possibly covered pixels is determined, and for each of those pixels, a coverage test is performed. To test the coverage, at a sample point in the center of a pixel a primitive shape's *edge equations* are evaluated. If it is determined that the primitive covers the pixel, the depth value of this sample is computed. For every pixel, there is a corresponding depth value stored in the *z-buffer* [Cat74]. Only if the depth value of a new sample is less than the already existing value in the z-buffer, its value is overwritten.

However, advanced optical effects like shadows, reflections, and refractions cannot be rendered correctly with rasterization because of two fundamental problems. First of all, the rasterization algorithm samples a scene

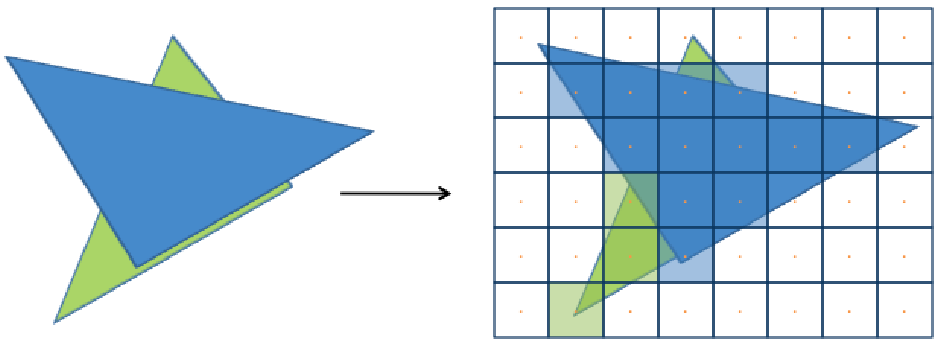

Figure 2.1: The rasterization algorithm projects sequentially all primitives onto the image plane. After the projection of each single primitive, its covered pixels on the image plane are determined. In order to allow a proper front-to-back rendering, for each pixel the distance to its current closest geometry point is stored. This distance is used to reject the attempt to overwrite pixels by primitives whose surface points are all farther away.

regularly on the image plane, but these samples for which the secondary effects should be evaluated are distributed irregularly in space. Currently, no hardware rasterizer is able to evaluate the visibility for this irregular distributed sample points. And second, the *direction* from which a sample should be generated cannot be considered. In short, with rasterization it is very complex to perform arbitrary *visibility queries*. A visibility query is defined as an operation that determines for a given origin and direction which geometry point is next, if any exist [Dur99].

The other rendering algorithm is *ray tracing*. In contrast to rasterization, ray tracing operates sequentially on the pixels. For each pixel, it can be computed independently what is visible using a *ray* concept. As will be discussed later, the ray concept allows for arbitrary visibility queries which makes pixel-accurate advanced optical effects conceptually simple.

2.2 An Outline of Basic Ray Tracing Concepts

The main notion of ray tracing is the ray $\mathcal{R}(t)$. A ray, in three dimensions, is a half-line parametrized by an origin $\mathcal{O} = (o_x, o_y, o_z)$, a direction vector $\vec{\mathcal{D}} = (d_x, d_y, d_z)$ and a ray distance parameter t such that any point p along the ray can be calculated by $p = \mathcal{O} + t\vec{\mathcal{D}}$.

To compute an image using the ray concept, a camera model generates for each pixel in the image plane a *primary* ray that is shot through it. In a naïve approach the rays are tested for intersection with all scenes primitives to determine the primitive and hit point with the shortest hit distance. To avoid the intersection test with **all** objects in the scene, spatial index structures can

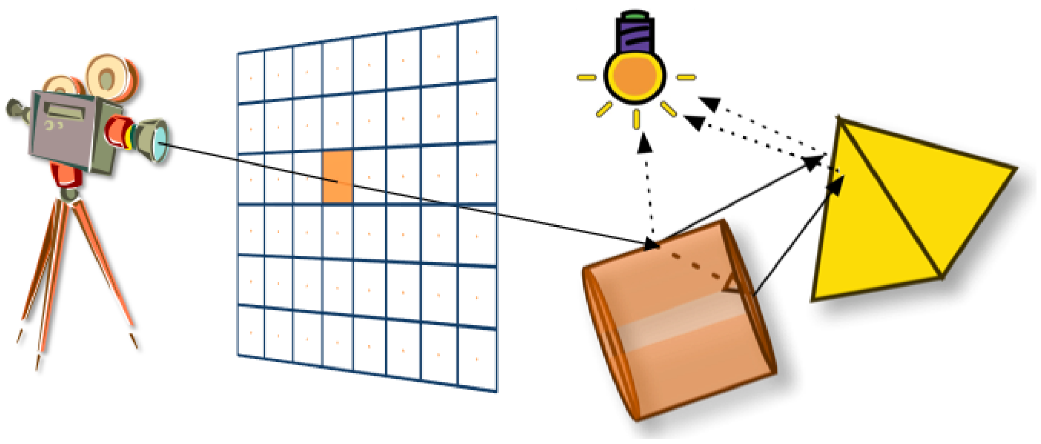

Figure 2.2: A recursive ray tracing example: A *primary* ray is shot through the center of a pixel in the image plane and the cylinder is found as the object with the closest hit distance. Since the cylinder uses a glass material description, *reflection* and *refraction* rays are generated. These rays hit the pyramid. Because of the definition of a light source in the scene, on all intersection points the objects surface shaders generate a shadow ray to determine if the hit points are lit or in the shade. Using the results of the recursively executed shaders, invoked from the primary and secondary rays, a pixel color is computed.

be built that allow a *traversal* with the ray and enumerate only those objects that are probably pierced by the ray (see Section 2.4). Index structures act as a database that take a ray as query key, and return the objects along the ray, not necessarily in a strict but better front-to-back manner, consecutively to a ray-geometry intersection routine until the closest hit point is found. Having found the closest point of intersection, a *surface shader* calculates a color that is assigned to the corresponding pixel. These shading computations can either use only *local* information that is available for the hit point, e.g. its color and shading normal, or query additional *global* information from the scene, e.g. if the hit point is lit by a light source.

All this shading computations depend on the used illumination and shading models as well as surface properties of the primitives. Based on these properties new *secondary* rays can be generated recursively within the surface shaders to query the global information that is needed for the shading computations (see Figure 2.2 for an example). This possibility of recursive ray generation allows also for complex lighting simulations, considering also indirect illumination from reflecting surfaces, resulting in photo realistic images. If only shading models are used that do not require any secondary rays the described algorithm is called *ray casting.*

From the above description the four core algorithms of ray tracing can be deduced: *ray generation*, *traversal*, *intersection*, and *shading.* A remarkable property of this algorithms is that they are completely independent. This means that the underlying algorithms and data structures can be easily replaced and different rendering primitives as well as shading models can be supported in one scene without any interdependencies. Only a common software interface is required that lets the different algorithms communicate, and share the required data.

2.3 Milestones in the Evolution of Ray Tracing

The very first ray tracing system was developed by Appel in 1968 [App68]. It was the first approach to calculate the visibility between two points with the concept of rays as described above. In this early work, only *planes* were supported as scene objects and a monochrome halftone shader was used to compute shades due to the absence of color output devices in these days. Nevertheless, this system produced for the first time images including accurate shadow effects from point lights.

In 1980, Whitted [Whi80] introduced the first fully recursive ray-tracing approach that allows to accurately render optical correct (mirror-like) reflections and refractions. In order to realize these effects he applied formulas from optical physics namely Fresnel's law for reflections and Snell-Descartes law for refractions. The actual render algorithm is split into two stages. In the first stage, a *ray-tree* is generated. This tree consists of the primary ray and all recursively generated secondary rays that are shot based on the surface properties at the hit points, but no actual shading computations are performed. The recursive ray generation stops automatically if no further object is hit, no further secondary ray is generated because of the surface properties at the hit point, or a specified recursion depth is reached. Afterwards, in the second stage, this tree is passed to a shader that traverses the trees nodes, evaluates its surface properties e.g. its specular color and shading normal, and computes based on all results the final pixel color. This approach was the first one that queried additional global information, using rays except for shadow computations, from a scene to increase the visual sensation of rendered images. Additionally, Whitted used also several primary rays per pixel to avoid aliasing artifacts.

So far the ray tracing algorithm was only able to render effects like hard shadows, reflections, and refractions. Cook et al. [CPC84] showed in 1984 that many more advanced optical effects are computable with a point sampling approach. Ray tracing is inherently a point sampling algorithm, but was until then only used to sample the image plane. Cook was the first one

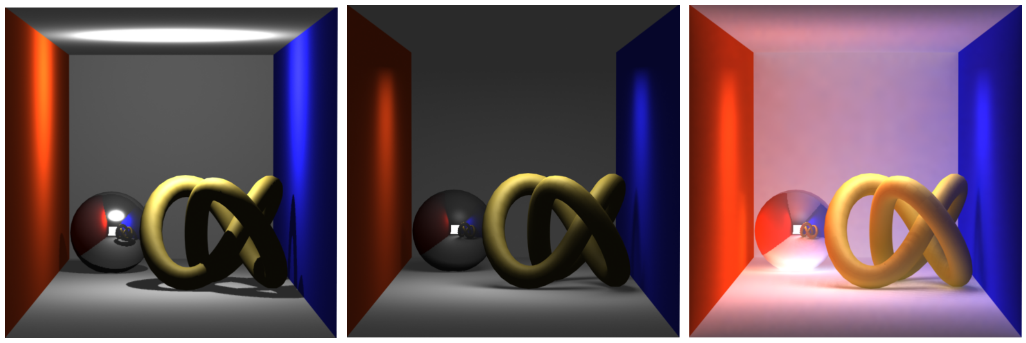

Figure 2.3: An example for the evolution of ray tracing image quality. On the left the scene is rendered in Whitted style with a single point light source. The middle image shows soft shadows sampled according to Cooks method. Finally, the right image shows a global illumination solution for the scene driven by Kayijas rendering equation. Please note how the realistic appearance increases from left to right.

who noted that additional rays can also sample motion, camera lenses, light sources, and shading functions. This sampling of the according functions allows then to render soft shadows, motion blur, depth of field, translucency, and glossy reflections. Using these effects enhances the realism of the rendered images significantly.

Although Cooks work was a big leap forward towards realistic images, ray tracing was still not able to render photo realistic images. This changed with the seminal work of Kajiya [Kaj86] in 1986. Kajiyas important contribution is the development of the *rendering equation*. His rendering equation describes the exitant radiance for any point in a particular direction and describes for the first time a generalized solution to the global illumination problem.

Considering light reflected from specular, glossy and diffuse surfaces allows to truly compute photo realistic images including indirect illumination as well as other optical effects like caustics [Jen01]. Caustics appear when light is concentrated on small diffuse surface areas by previous specular reflections. Kajiya presented also a method for an approximate solution of the rendering equation because in the general case, the rendering equation cannot be solved analytically and has to be approximated numerically, e.g. by using Monte Carlo techniques like point sampling.

The basic idea is to recursively forward a **single** ray from hit point to hit point until a light source is hit, or a defined *path length* is reached. At the light source, the light contribution of this *ray-path* can be computed for a pixel. However, many rays per pixel have to be shot to obtain a smooth, (almost) noise free image as many ray-path do not hit a light source and such do not contribute radiance to the pixel. This approach of "shooting a ray-path" to approximate the rendering equation is called *path-tracing*. Figure 2.3 compares the visual appearance of a test scene rendered with Whitted's, Cook's, and Kajiya's method.

2.4 Realtime Ray Tracing Techniques

As described above, ray tracing image synthesis algorithms enhanced significant over the years up to global illumination simulations. Unfortunately the time to compute ray traced images at times of Whitted and Cook were in the order of several hours, up to days, and for a long time ray tracing was considered only as a high-quality offline rendering algorithm. Starting in the 1990s [Muu95, PMS$^+$99], researchers started to investigate concepts that are necessary to speed up ray tracing to interactive or even realtime performance. Three important concepts for realtime ray tracing are *parallelism*, *coherence*, and *acceleration structures*. Two important use cases for these acceleration structures are: the reduction of the required ray-geometry intersection tests, and the reduction of the computational costs for a single ray-surface intersection. Typically these data structures are built in a preprocess before rendering takes place and feature the classical performance-memory trade-off. In the next sections all these concepts will be discussed in more detail.

2.4.1 Parallelism and Coherence

A property of the ray tracing algorithm is that every computed sample on the image plane is completely independent from each other. For this reason, ray tracing is sometimes called an *embarrassingly* parallel algorithm. In multicore/multiprocessor environments, this fact can be exploited by assigning different sets of samples to the processing elements and later combine the parallel computed samples to a final image. Parker et al. [PMS$^+$99] showed in 1999 that realtime ray tracing performance can be achieved using large supercomputers, e.g. an SGI Onyx with more then 128 processors processors, even for complex scenes with non-trivial lightning simulations.

Another possibility to speed up the rendering performance is to exploit *coherence*. An obvious example for coherence in image synthesis is *image space* coherence, i.e. neighboring pixels display the same geometric object. A definite definition of the term coherence in the context of ray tracing cannot be found in the literature [Ben06]. But loosely speaking it means that: *similar* rays will likely execute similar code and access similar data. The similarity of rays is very dependent on their origins as well as directions. Rays that have a common origin and similar direction, as primary rays of neighboring pixels, have a very high coherence whereas rays with origins that are spread around the scene and with very different directions have a very low coherence.

Wald [Wal04] and Benthin [Ben06] showed that exploiting coherence using Single-Instruction-Multiple-Data (SIMD) extensions of modern CPUs can speed up the rendering performance by a factor of two to three by tracing four rays in a data-parallel fashion. A packet size of four rays is chosen because current SIMD architectures allow to operate on four 32 bit data values in parallel [AMD, Int02b]. The expected performance gain of a factor of four cannot be achieved in general due to some overhead, e.g. because of masking operation for rays that already terminated. Exploiting SIMD reduces also the required memory-bandwidth and cache miss rate since fewer data accesses are necessary. Additionally, the penalty costs for a memory access can be amortized to a fourth e.g. when a requested data word is used in all four slots of a SIMD register. In their work they also showed that SIMD extensions can be used for all four core algorithms (ray generation, traversal, intersection, and shading) as well as complex global illumination computations [BWS03a].

A recent approach to exploit ray traversal coherence more efficiently was presented by Reshetov et al. [RSH05] in 2005. Their approach is based on the observation that coherence while ray traversal is not constant. Most ray tracing implementations use hierarchical index structures (see Section 2.4.2) to speed up first-hit calculations. It can be seen that the coherence up in a hierarchy for ray packets is very much higher then in lower levels. This led to the idea that it is reasonable to start traversal with a very large ray bundle, e.g. with 64×64 rays, and during traversal split the bundle into smaller subbundles as coherence decreases. In fact, Reshetov starts with a large ray bundle and calculates the deepest node in the hierarchy that covers all primitives that can be intersected by the ray bundle. Afterwards the ray bundle is split and traversal starts again for the subbundles at the previously calculated *entry-point* for the new ray bundles. This refinement of ray bundles is repeated until the entry-point is at the lowest level of the hierarchy or a ray bundle contains only four rays. To keep the computational costs low, only the corner rays of a ray bundle are considered for traversal. Using this technique, ray traversal performance can be accelerated by an order of magnitude.

As ray traversal coherence for primary rays can be exploited efficiently even for large ray bundles, it is much more difficult for secondary rays and further processing steps like ray-geometry intersections and shading. In order to increase the coherence of secondary rays, sets of secondary rays can be reordered, e.g. according to their directions and origins, into more coherent ray sets that are then traced independently [PKGH97, NFLM07, MMAM07]. But until today these approaches do not deliver the expected performance gain due to high reordering costs.

Intersection test costs for large ray bundles can be reduced by intersecting first the corner rays of the ray bundle with the primitives. If the corner rays hit the same primitive, the intersection points for *inner* rays can be linearly interpolated – assuming that the primitives are convex [WBS07].

2.4.2 Hierarchical Index Structures and their Construction

A naïve ray tracing implementation would intersect a ray with all primitives of a scene to determine the closest hit point. For scenes with many primitives this is not feasible as rendering times would be far from interactive. In this section data structures are discussed that are suitable to reduce the number of required ray-geometry intersections by organizing the geometry in hierarchical spatial structures like the *bounding volume hierarchy* (BVH) [RW80, Whi80] or kD-trees [Kap85, Jan86]. Although there are many other data structures – like grids [AW87], bounding interval hierarchies [WK06], bkd-trees [WMS06], etc. – that can be exploited for this task, only BVHs and kD-trees are discussed here and used throughout this thesis, because they typically offer the best or at least competitive performance results on commodity PC hardware – when built properly [Hav01a].

The very basic idea of these hierarchical data structures is to distribute the scenes primitives recursively into two sets; resulting in a binary tree structure with only a few primitives left per leaf node. In doing so, the BVH and kD-tree follow two fundamental different approaches. A kD-tree is build by splitting a given space into two half-spaces, divided by an axis aligned plane, and distributes the objects into the half-space they belong to. Objects that overlap both half-spaces will be inserted into both. This process is recursively repeated until some termination condition is reached (see below).

A BVH does not split the objects *space*, but splits the set of objects into two disjoint sets which yields in turn two enclosing volumes, one for each set – which can overlap or be completely disjoint. After each split the exact bounds, of each subset, are computed and stored in the respective node. The decision, which primitive belongs to which set, is here also based on an axis aligned splitting plane. For objects that lie on the splitting plane the objects median can be used to evaluate the best matching set it belongs to. The enclosing volumes can be arbitrary defined but in practice often axis aligned boxes are used since they can be very efficiently handled in the traversal stage (see Section 2.4.3). Since in a BVH a primitive always belongs to only one split subset, the tree has usually less nodes than a kD-tree for the same scene. Additionally, a kD-tree will typically have some empty leaf nodes whereas a BVH does not. Nevertheless, both acceleration structures have a

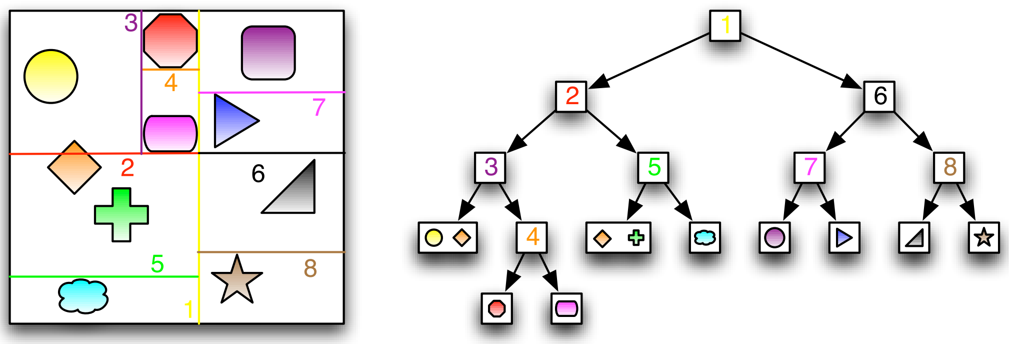

Figure 2.4: A simple kD-tree construction example: On the left, the recursive space subdivision with splitting planes. The colored numbers indicate at which step in the process the corresponding splitting plane is inserted. On the right, the resulting binary tree structure. Each internal node in the hierarchy shows the number of its corresponding splitting plane.

construction time complexity of $\mathcal{O}(n \log(n))$ which can make this procedure computational expensive. Figure 2.4 shows a simple kD-tree construction example.

Build Strategies

An unanswered question is now when a space, or set of objects, is split by an axis aligned plane what the orientation (x, y, or z axis) and the position of this splitting plane along the split axis is. Probably the simplest method to choose the orientation of the splitting plane is to cycle through the dimensions x, y, and z. Two simple strategies can then be used to determine the splitting plane position along the splitting axis. First of all, the splitting plane can be put in the middle of the space. Second, rather then cutting the space in the middle, another strategy is to position the splitting plane at the location where the number of objects on both sides is balanced (median).

Nevertheless, currently the best known method for calculating the splitting dimension and position is the so called *surface area heuristic* (SAH) proposed by Booth et al. [MB90]. Using the SAH can lead to rendering performance improvemtens up to a factor of two [Wal04] compared to other strategies. The SAH evaluates at each possible split position – in all dimensions – a cost function:

$$C = C_T + C_I \left(N_L \frac{SA(V_L)}{SA(V)} + N_R \frac{SA(V_R)}{SA(V)} \right) \tag{2.1}$$

to determine the split position where C is minimal. C_T describes the computational costs for a traversal step and C_I for intersections respectively. These costs have to be measured for each actual implementation since they are very dependent on the used algorithms for traversal and intersection, (compiler)-optimizations and hardware architecture. N_L and N_R are the number of objects in each of the split partitions. Finally, the function SA determines the surface area of the complete volume V as well as both split partition volumes V_L and V_R. It can be shown that the minimum C lies always on a vertex position (for triangles) and thus only at those positions equation 2.1 has to be evaluated.

A termination criterion to stop the recursive splitting process is given e.g. when the total intersection costs are smaller than the traversal costs. Additional criteria that are often used are a maximum node depth in a tree or a maximum number of objects in a leaf node. The SAH leads to a *good* tree structure because it tries to minimize for arbitrary rays the total computational costs for traversal and intersections. Although the SAH turns the tree construction process into a *greedy* [1] algorithm that could lead to non-optimal results, no better algorithm is currently known; except the brute force variant that tests **all** possible trees for best performance which is impractical for non-trivial scenes.

A recently published new strategy [WK06, WMG$^+$] for a fast BVH construction is similar to the above mentioned split-in-the-middle strategy. But rather than computing after each split new exact bounds for the subtrees the current bound is simply divided in the middle such as the kD-tree *spatial* subdivision scheme. When the tree build is finished a recursive update process computes for all tree nodes the correct bounds.

2.4.3 Ray-Traversal Algorithms

The last section described briefly how a BVH or a kD-tree can be built. Now it will be discussed *how* these data structures can be *traversed* to only perform intersection tests with primitives that are potentially hit by a ray. Many different traversal algorithms exists for kD-trees and BVHs, but here only those will will be discussed that are used in this thesis.

kD-Tree Traversal

As mentioned above, a kD-tree is a binary tree with each node having either no (leaf node) or two descendants (inner node). Each internal node stores a pointer to its children, the splitting plane dimension and position, as well a flag that indicates whether it is a leaf node or not. It is possible to use only one child pointer if both children are stored side by side in memory. A

[1] A greedy algorithm always makes the choice that looks best at the moment [CLRS01]. For each splitting position the optimal **local** solution is computed with the goal to achieve a globally optimal solution. However, this goal cannot always be achieved.

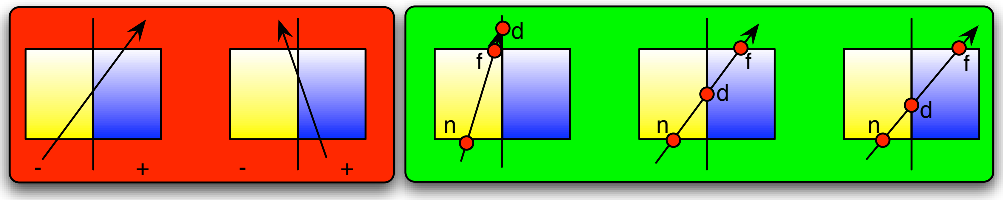

Figure 2.5: Single ray kD-tree traversal exemplified. In the red box the correlation between a rays direction and the near/far half-spaces is shown. It can be seen that the decision which half-space is the near or far one is solely dependent on the rays direction. The green box shows the three standard kD-tree traversal cases. If $d > f$ ($n > d$), only the near (far) half-space has to be traversed. If $n > d < f$ both half-spaces have to be considered. n is the distance to the entry point of the ray, d the distance to the splitting plane, and f the distance before the ray leaves the bounding box.

leaf node stores the number of contained triangles as well as a pointer to the triangle list. All this information, for both internal and leaf nodes, can be stored in a single packed representation with eight bytes [Wal04].

A single traversal step in a kD-tree is computational very cheap. It requires only the distance calculation of a ray's origin to a nodes split plane, and at most two conditionals. The the two conditionals determine if either the *near* or the *far* node has to be traversed, or both. It can be shown, that the decision, which of the children's nodes is near (far) is solely dependent on the ray direction along the split axis (Figure 2.5 red box). A remarkable property of the kD-tree is that it allows a strict front-to-back traversal. That means that once a leaf node is found which contains a triangle that is hit by a ray, traversal can immediately stop, because no other leaf node can contain a triangle with a closer hit distance. In the following the decision rules for single ray, coherent SIMD traversal, and frustum traversal are detailed. For further information including source code please refer to [Wal04, Ben06, Hav01a].

Single Ray Traversal: The single ray traversal begins by clipping the ray against the bounding box of the complete scene. This yields ray distance parameters near, n, and far, f, to the entry and exit point with the scenes bounding box – assuming that the ray hits the box at all. These initial parameters are required to start the traversal procedure. After calculating the ray distance d to the splitting plane, three traversal cases have to be distinguished (see Figure 2.5 green box). Either the ray traverses only the near (far), or both half-spaces. A ray traverses only the near (far) half-space when $f < d$ ($n > d$), and both in any other case.

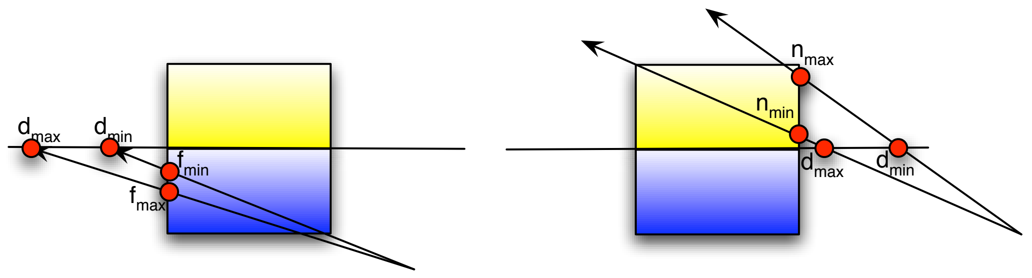

Figure 2.6: The frustum ray segment kD-tree traversal method. For traversing a complete frustum, containing potentially many rays, only a **single** distance interval $[n_{min}, f_{max}]$ has to be considered. If $d_{min} > f_{max}$ only the near half-space has to be traversed. In the case that $d_{max} < n_{min}$ only the far half-space. All other cases require the traversal of both half-spaces.

SIMD Ray Packets: Considering now a SIMD packet of 4-rays, the same rules for the traversal can be used – when all rays share the same origin. But rather comparing only single values now four values are compared at the same time using SIMD extensions. Identical to the single ray traversal, all rays in the packet traverse only the near (far) half-space when $\forall i \in [0,3] f_i < d_i$ $(\forall i \in [0,3] n_i > d_i)$, and both in all other cases. As said above, the decision which half-space is near (far) depends on the ray directions. If the directions in a packet differ, either the packet can be split and single ray traversal is used, or the common origin of the rays is used to guarantee a consistent front-to-back traversal order. All rays traverse first the half-space that includes the origin. In the case that both half-spaces have to be traversed, it is possible that not all rays actually pierce both half-spaces. To avoid unnecessary computations these rays are just *turned-off* by using an active-ray mask. Similarly for rays that have already found their closest hit, they will just be deactivated such that they cannot influence the traversal decisions anymore.

Frustum Traversal: Finally, the last kD-tree traversal scheme that is briefly touched in this introduction is the frustum traversal (see Figure 2.6). Similar to the packet traversal case, a frustum consists also of four rays. But now, these rays are only the corner rays of a (possibly) larger ray bundle e.g. with 8×8 rays. Interestingly enough, it is possible to traverse a complete ray bundle by just looking at a single distance interval $[n_{min}, f_{max}]$. If the $d_{min} > f_{max}$ only the near half-space has to be traversed. When $d_{max} < n_{min}$ only the far half-space. In all other cases both. The algorithm starts again by clipping the frustum rays against the scenes bounding box. To obtain initial values for n_{min} and f_{max} the minimum and maximum of all n_i and f_i can be determined. d_{min} and d_{max} can also be computed without looking at all individual rays in the bundle. Before traversal starts all minimum $min\_dir_i = \max(\forall frustum\_dirs_i)$ and maximum $max\_dir_i = \min(\forall frustum\_dirs_i)$ frustum directions, with $i \in \{x, y, z\}$, are calculated. These *extreme* directions can then be used to compute d_{min} and d_{max}.

BVH Traversal

A BVH is a binary tree where each node has either no, or two descendants. Contrary to the kD-tree, a BVH *cannot* guarantee a strict front-to-back traversal. The reason is that a BVH may have overlapping boxes and thus it cannot be known in which box the first hit can be found. This implies that traversal cannot just stop after the first leaf node is found which contains a triangle that intersects the ray and that there is an traversal overhead as the ray may traverse parts that it would would not traverse otherwise.

Single Ray Traversal: Typically, when single ray traversal is used, a BVH node stores an axis aligned bounding box, a leaf indicator, and a single pointer to its children or to its primitives – plus the number of primitives – when it is a leaf node. In a traversal step, the entry distance to its children's bounding boxes are computed, and the closest one is traversed first. The termination criterion for a ray is reached when the entry point of the next node is farther away as the current closest hit position with a primitive. Since the costs of a single traversal step are much larger compared to a single kD-tree traversal step, single ray BVH implementations are usually slower, at least on CPUs.

SIMD Traversal: In contrast to single ray traversal, SIMD traversal implementations do in general not intersect the bounding boxes of a node's children but the node's own bounding. The reason is that for the rays in a SIMD packet, not all rays will have the closest hit with the same children's box. Thus the particular distances cannot be efficiently used to determine the best traversal order. For that reason, another strategy is used which *approximately* traverses the boxes in distance order [Mah05] using the dimension of the clipping plane (determined while BVH construction), and the ray direction's signs [WBS07] – in the same manner as a kD-tree decides which descendant is the near-son. Many implementations use for convenience just the sign bits of the first ray in the packet and precompute before traversal starts for each dimension the near-son index. The less the bounding boxes of the BVH overlap the better this heuristic works.

Extended SIMD Traversal: As the traversal costs even for SIMD ray packets are still much larger than for kD-tree traversal, Wald et al. [WBS07] propose a new algorithm to amortize the expensive ray-box intersection tests

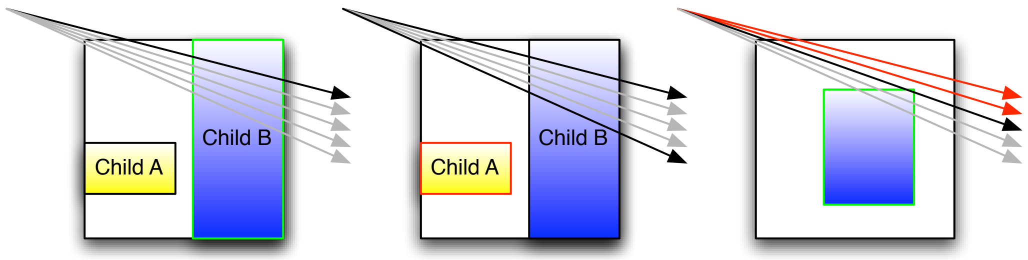

Figure 2.7: First-active descent, frustum test, and active ray tracking: Given a BVH node, the first *active* ray in the packet against the bounding box is speculatively tested. If it hits it can immediately be descended (left). If this test fails, a frustum test to reject nodes completely outside the frustum is performed (center). If neither of these tests prove successful, all rays sequentially in a packet are tested until one hits; rays that missed are deactivated for future traversal steps (right).

over larger ray bundles e.g. with 8×8 rays. The core idea is to split a traversal step into three individual tests: At first, the first active 4-ray packet is tested for intersection with the nodes bounding box. If this test is successful, immediately the next node to be traversed can be computed without testing all other active ray packets in the complete bundle. If this tests fails, the frustum rays of the bundle are tested with the bounding box to decide if the frustum hits the bounding box. If this test also fails, all active ray packets are consecutively tested for intersection until a packet is found that hits the box. For subsequent traversal steps, this packet is then the first active packet (see Figure 2.7).

2.4.4 Fast Ray-Triangle Intersection Tests

Today, the most common used graphical primitive is the triangle and realtime ray tracing engines use optimized ray-triangle implementations that can lead to a performance improvement of up to a factor of two [Wal04].

A triangle is simply defined by three vertices $v_i, i \in 1, 2, 3$ with $v_i = (x_i, y_i, z_i)$. To compute **if** and **where** a ray hits a triangle a system of linear equations can be set up i.e. $\mathcal{R}(t) = \alpha v_1 + \beta v_2 + \gamma v_3$. t is the hit distance along the ray, and α, β, γ are the barycentric coordinates of the hit point. A ray hits a triangle if $t \geq 0$, $\alpha + \beta + \gamma \leq 1$, and $0 \leq (\alpha, \beta, \gamma) \leq 1$.

In order to derive an efficient solution it is common to reformulate this system of equations to an *edge*-based form resulting in $\mathcal{R}(t) = v_1 + \alpha \vec{e_1} + \beta \vec{e_2}$ with $\vec{e_1} = v_1 - v_0$ and $\vec{e_2} = v_2 - v_0$. Möller et al. [MT97] rewrite this formula as matrix product:

$$\mathcal{O} - v_0 = \begin{bmatrix} \vec{\mathcal{D}} \; \vec{e_1} \; \vec{e_2} \end{bmatrix} \begin{bmatrix} t \\ \alpha \\ \beta \end{bmatrix}, \tag{2.2}$$

and apply then Cramer's Rule to compute t, α, and β.

Badouel [Bad90] takes a different approach. Rather than computing t, α, and β in one step, he first computes t by intersecting the ray with the infinite plane the triangle is embedded in. This can be simply done by e.g. substituting the ray equation $p = \mathcal{O} + t\vec{\mathcal{D}}$ into the plane equation $(p - v_1) \cdot N = 0$, with N being the triangles geometry normal. Having now t, and thus the hit point coordinates, Badouel reduces the equation $p = v_1 + \alpha\vec{e_1} + \beta\vec{e_2}$ to two dimensions by projecting the triangle on an axis-aligned plane. This allows to perform all following computations in 2D. In order to avoid numerical issues, the projection plane is chosen such that the projected area of the triangle is as large as possible.

Wald [Wal04] showed that many calculations in Badouels approach are redundant and can thus be precomputed and reused. In his approach, for each triangle a structure is precomputed that contains values that are constant per triangle for computing t, as well as values that can be used to compute α, and β. Although this structure increases the memory consumption, cache utilization does not suffer. On the contrary, cache efficiency is increased because all relevant data that are needed for intersection are stored in one consecutive block of memory, e.g. one cache line, and thus no random memory access occurs. Whenever in this thesis a ray-triangle intersection test is needed a variant of this approach is used. For more details and a thorough discussion please refer to [Wal04].

Chapter 3

Ray Tracing for Computer Games

This chapter comments briefly on the current state-of-the-art in ray tracing and exemplifies why ray tracing should be used for the development of future computer games. It will be also discussed what kind of software and hardware support is necessary to make ray tracing happen in future computer games.

3.1 General Discussion

Computer games are the single most important force pushing the development of parallel, faster, and more capable hardware. Some of the recent 3D games (e.g. *Elder Scrolls IV: Oblivion* [Bet05]) require an enormous throughput of geometry, texture, and fragment data to achieve high realism. They increasingly use advanced and computationally costly graphics effects like shadows, reflections, multi-pass lighting, and complex shaders. However, these advanced effects become increasingly difficult to implement due to the fundamental limitations of the rasterization algorithm, its inability to perform recursive visibility queries from within the rendering pipeline. This results in a number of significant problems when trying to implement advanced rendering effects. These limitations will be analyzed in more detail in Section 3.2.

Ray tracing, on the other hand, has several advantages and avoids many of these limitations (also discussed in Section 3.2). It is, for example, specifically designed to efficiently answer exactly these recursive visibility queries, which enables it to accurately simulate the light transport and the appearance of objects in a scene. However, ray tracing had been much too slow for interactive use in the past. Due to significant research efforts in recent years, ray tracing has achieved tremendous progress in *software ray tracing* performance [WSBW01, RSH05, WBS07, WIK$^+$06] to the point where realtime frame rates can already be achieved for non-trivial scenes on standard CPUs and at full screen resolution (see Table 3.1).

Table 3.1 compares the rendering performance of several realtime ray trac-

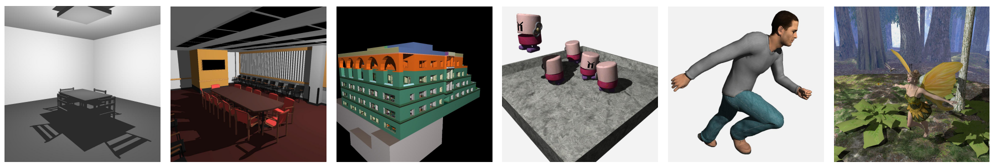
Figure 3.1: Several ray tracing benchmark scenes for performance comparisons. From left to right: ERW6, Conference, Soda Hall, Toys, Runner, Fairy.

ing implementations, namely the original OpenRT system [WBS02], multi-level ray tracing (MLRT) [RSH05] both using kD-trees as spatial index structures, and recent implementations with Bounding Volume Hierarchies (BVH) on CPUs [WBS07] as well as GPUs [GPSS07], and Grids [WIK$^+$06]. These numbers give an overview of the ray tracing performance that can be achieved in software, but it is important to note that these systems vary significantly in their feature set and thus are not directly comparable. Images of the used test scenes are shown in Figure 3.1. This speedup in software was possible due to a number of algorithmic improvements across the entire ray tracing pipeline, beginning with the spatial index structures (see Section 3.5 for more details) and associated traversal algorithms [Hav01a], ray-primitive intersection tests e.g. for triangles [Wal04] or Bézier surfaces [BWS06a], and shading calculations. These algorithmic improvements have been augmented and

Scene	#Triangles	OpenRT 2001	MLRT 2005	BVH 2006	Grid 2006	GPU BVH 2007
ERW6	800	2.3	50.7	31.3	18.3	36.0
Conference	280k	1.9	15.6	9.3	4.0	19.0
Soda Hall	2.5M	1.8	24.0	10.9	7.4	16.2
Toys	11k	–	–	21.9	20.0	–
Runner	78k	–	–	14.2	13.1	–
Fairy	180k	–	–	5.6	3.1	–

Table 3.1: Performance comparison of several ray tracing implementations measured in frames per second (fps). Note that these systems are not directly comparable due to their highly varying feature set and exploited hardware platforms: some even support dynamic scenes while others do not work well for secondary rays. All numbers have been measured with a simple diffuse shader at a resolution of 1024 × 1024 pixels on a single high-end CPU core except the GPU numbers which are measured using an NVidia G80 graphics board.

carefully tuned with optimizations for today's CPU architectures [Ben06].

Due to its high degree of parallelism the ray tracing performance benefits directly from the current trend towards symmetric and asymmetric multi-core CPUs and affordable multi-processor systems. In addition, dedicated ray tracing hardware has been developed [SWW$^+$04, WSS05] that are programmable and support fully dynamic scenes [WMS06].

These developments on the algorithmic as well as on the software and hardware implementation side are promising to bring ray tracing performance to a level where it becomes interesting for computer games. In the following sections this space will be explored by looking at the potential benefits of using ray tracing in games (Section 3.2), several prototypes of ray-tracing based games (Section 3.3) are analyzed, the importance and the different requirements of ray tracing for APIs (Section 3.4) are stressed and recent advances in the support of dynamic scenes (Section 3.5) are reviewed. Finally, the different options of improving performance through better hardware support will be analyzed and compared.

3.2 Advantages of Ray Tracing for Games

Before implementations and other details of ray tracing are discussed it is important to first take a look at the advantages and opportunities this technology may offer for games.

3.2.1 Recursive Visibility Queries

Due to the fundamental feed forward structure of the rasterization algorithm the complexity of implementing advanced features has increased significantly. One indication of this is the large number of publications about reducing shadowing artifacts in rasterization and the fact that games sometimes still need to implement multiple shadow algorithms in order to handle each gaming situation [Hur05].

In this context it is important to note, that most computations that occur during rendering depend on the *locally visible* part of the scene. This includes, for example, the surfaces visible from a camera, light sources illuminating a point, or other surfaces visible in reflections or causing indirect illumination. Such visibility queries sample the global *visibility function* which maps some local coordinates (typically position and direction) to global coordinates (i.e. a point on *some* other surface). Interestingly enough, these visibility queries are exactly what recursive ray tracing is designed to compute efficiently.

Rasterization, however, does not allow for any recursive visibility query from within its pipeline. To support such computations at all, it must fall back to precomputing the data *in local coordinates* in form of the well-known shadow, reflection, or other maps. However, there are two major issues inherent in this approach: inefficiency and sampling artifacts.

Inefficiencies arise because in general the entire visibility function for a point or an object must be precomputed even if later only a small subset may be relevant (e.g. not everything locally visible will contribute to a reflection). This is a fundamental problem because the final queries are not known in advance (otherwise, the result of the previous visibility queries that caused these queries in the first place would already be known).

Typically discrete maps are used to represent relevant parts of the visibility function. Thus, aliasing artifacts are unavoidable because this function is generally continuous and not band-limited.

The design of the rasterization algorithm causes yet more problems with visibility queries. The algorithm is designed to be efficient only for a very large set of very regular queries (i.e. millions of rays all starting at a single origin and going through the uniform grid of pixels on an image plane). However, the visibility queries that occur during rendering tend to be rather sparse and irregular. An example are the few reflection rays of a small curved reflector. Furthermore, this rasterization process is driven directly by the application in an imperative way ("draw these triangles to the (frame) buffer in this way"). However, the need for recursive visibility queries arises from shading computations within the pipeline on the graphics chip. This requires some form of feedback from the hardware through the driver back to the application running in a totally different (user) context. This significantly complicates the process and introduces large latencies.

It is interesting to note that due to the above problems ray tracing is increasingly being used as a supplementary algorithm within rasterization based techniques. Examples are displacement mapping [WTL+04], approximate refractions on the GPU [Wym05], or ray casting through volumetric data [KW03b] (also see Section 3.6.2 regarding the implementation of ray tracing on GPUs). In this chapter, it is argued that it may be interesting for games to explore ray tracing as the *primary rendering engine* instead of just an add-on.

3.2.2 Plug'n'Play for Geometry and Shading

Ray tracing closely matches the physical process of light transport in the real world. This has several important implications also for the design of games when utilizing ray tracing. The most notable advantage is what is called "Plug'n'Play" for geometry and shaders (see also Section 3.4).

The recursive ray tracing method can support a wide variety of geometric primitives as long as efficient intersection algorithms are known and the objects can be inserted into spatial indices. Beyond triangles, realtime algorithms have been published for (trimmed) splines surfaces [BWS06a, GA05],

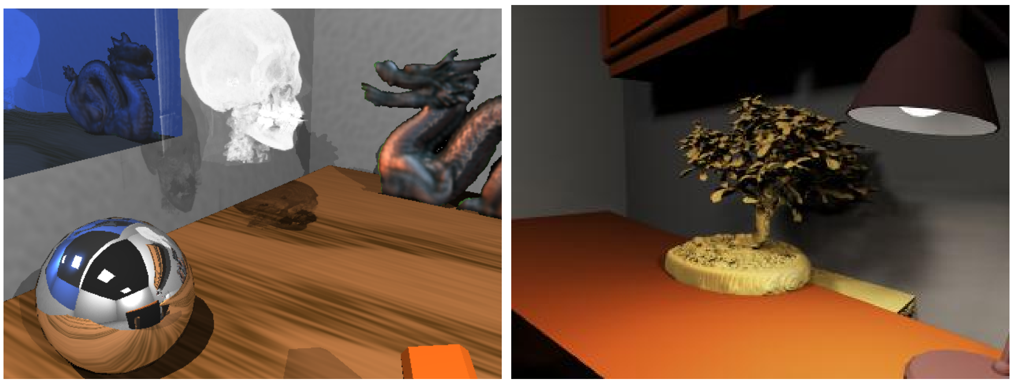

Figure 3.2: Left: an office scene with polygonal surfaces, volume data, and a light field object. Note how all optical effects work as expected: For example, the volume (skull) casts semi-transparent shadows on procedurally textured surfaces; the light field (dragon) is also visible through the bump-mapped reflections on the mirror; and all these effects are again visible in the mirror sphere. Right: a volumetric Bonsai tree in a polygonal environment with indirect global illumination and soft shadows. No additional code is required to handle the complex interaction of light between all these different geometry representations.

subdivision surfaces [BWS06a], point based surfaces [WS05], and iso-surfaces of volumetric data [WFM+05, MS06a]. Even more algorithms are known from off-line computation that may also be accelerated for realtime use. The RPU ray tracing architecture [WSS05] even supports programmable and hardware accelerated intersection computations.

Because they all support the same simple interface towards rays, *all* of these representations can be freely mixed in the same scene. All interaction between different geometry objects such as reflections or shadows work seamlessly as is demonstrated in Figure 3.2. No special support is needed by the application, which is in stark contrast to rasterization, where specialized rendering algorithms must be integrated for every type of primitive. It is well-known that these algorithms are generally not orthogonal to each other and often interfere also with shading algorithms.

For shaders the situation is similar. Shaders describe the material properties for ray tracing and specify *locally* how light interacts with the surface they are applied to. However, as discussed above, they may also query about their global environment by recursively shooting more rays. The *global* appearance is then achieved automatically *without further participation* by the application. Seemingly difficult effects such as complex inter-reflections, multiple refraction, or self shadowing are straightforward to achieve in this context (see Figure 3.3). Also no explicit depth sorting of transparent polygons is needed as with rasterization, because the ray tracing algorithm is guaranteed to visit all surfaces in the correct order.

Together, this allows for full "Plug'n'Play" where shaders can be freely assigned to geometry, which may then be freely combined with other objects and their shaders. This simplifies both the development of the graphics engine in a game and the creation of game content. The content designers are no longer constrained by sometimes obscure technical limitations. Many advanced effects that were difficult to achieve with rasterization (such as simple shadows) come naturally with ray tracing.

3.2.3 Scalability in Scene Size

The scene complexity in games is increasing at an amazing rate and already reached the point where the basic approach (sequential brute-force rasterization with z-buffer) cannot handle these scenes any more. Increasingly techniques like level of detail (LOD) [LWC$^+$02], spatial indices like BSP-trees [FKN80], portals [LG95], and precomputed potential visible sets (PVSs) [ARFPB90] are used to reduce the number of triangles that must be processed by the hardware.

It is interesting to note that many of these techniques require data structures that are almost identical to those used within ray tracing. However, these data structures are usually built and maintained on the application side instead of built transparently into the rendering engine itself. As a result, the granularity of these approaches is usually much coarser than for ray tracing.

Ray tracing is output sensitive by design, i.e. it only ever processes (or even loads) data touched by rays that are known to contribute to the image. The typical spatial index structures are hierarchical and lead to an average computational complexity of $\mathcal{O}(\log(n))$ for ray shooting for scenes with n primitives. This allows for ray tracing of scenes with billions of triangles in realtime as is shown in Figure 3.8.

3.2.4 Physically-Based Global Lighting Computations

Another major advantage of recursive ray tracing is that it can directly be mapped to the process of simulating the transport of light particles through a scene. Here each ray represents some portion of energy that travels through space until scattered at some surface (or even within participating media). These events may then recursively spawn new rays. This directly allows for rendering reflections and refractions even via highly complex light paths as seen in Figure 3.3. All these lighting computations are typically performed with floating point precision and thus directly support High-Dynamic-Range (HDR) rendering.

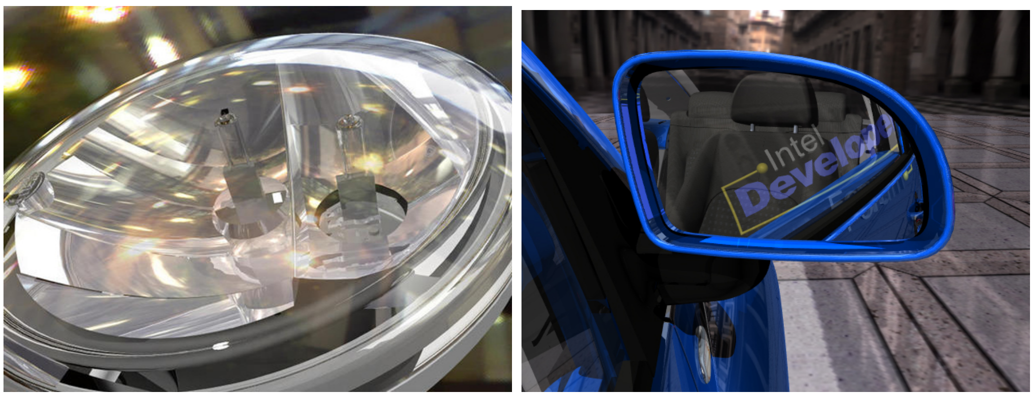

Figure 3.3: Left: complex light path with up to 25 levels of recursive reflection and refraction for the car headlight with highly curved surfaces. Up to 50 rays per pixel need to be traced to achieve an adequate realism. Right: An example from the industrial use of realtime ray tracing simulating important multiple reflections for design reviews.

The subtle variations due to indirect illumination significantly increases the realism of game environments, as it contains information about the spatial proximity of objects that the human visual systems interprets quickly and subconsciously. Indirect illumination at some point is caused by light that bounced off of (all) other visible surfaces in the scenes, and may recursively involve other non-visible surfaces as well.

This illumination technique has been widely applied in commercial games but has been limited to static and precomputed results because of the huge computational cost. Typically radiosity methods [CW93] are used to compute the indirect illumination, which is then stored as vertex colors or light textures. One increasingly popular method also is precomputed radiance transfer (PRT) [KSS02, SKS02], which allows changing the illumination of static objects through a distant environment map. Newer extensions [SLS05] allow also the use of PRT for locally deforming objects.

An important general technique to efficiently compute indirect illumination is Instant Radiosity [Kel97]. In a first pass this technique places secondary light sources in the scene through a random process. In the second pass these light sources are used to illuminate the scene using normal shadow computations. This approach directly maps to an efficient implementation using fast ray tracing [WKB$^+$02, BWS03b] that already runs in realtime on a small set of CPUs.

Another indirect illumination effect are caustics caused by the concentration of light due to reflections or refractions (see Figure 3.4). Caustics can provide important visual clues, especially with transparent objects such as

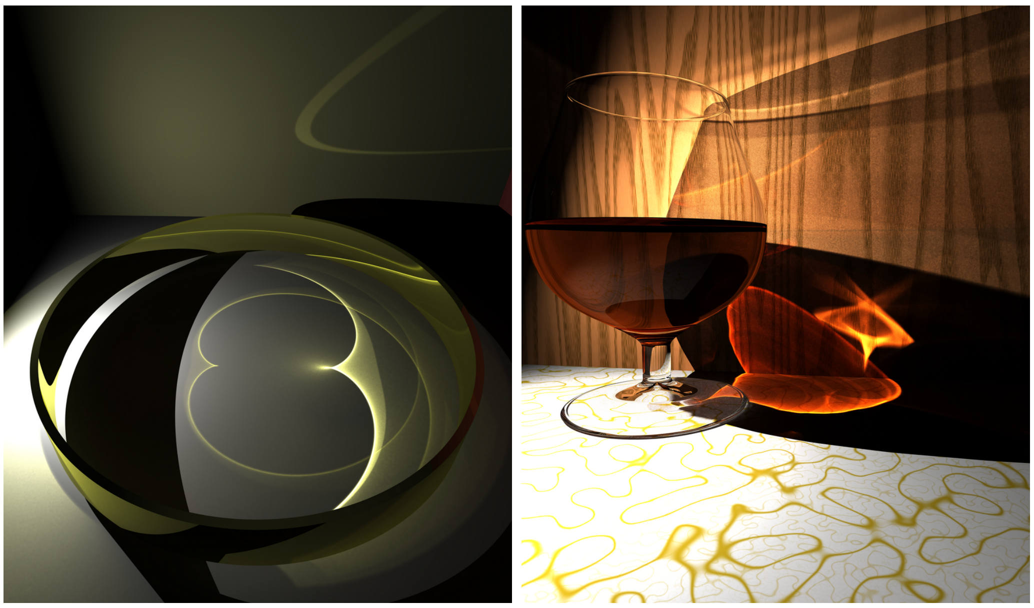

Figure 3.4: Caustic effects caused by multiple reflections or refractions are well handled by photon mapping based on ray tracing. This images can be rendered at approximately 15 fps on a cluster of PCs with a resolution of 640 × 480 pixels.

glass and liquids. These scenes would look highly unnatural without their caustics illumination effects.

Photon Mapping [Jen01] is considered the only robust and efficient technique for computing caustic effects. In a first pass it distributes large number of "caustic" photons in the scenes via specular reflection of refraction. In the second pass it then uses the density of photons at a point to compute the caustic illumination. This method can again be mapped directly to ray tracing (see Figure 3.4) and reaches near-interactive frame rates even on a single CPU [GWS04].

While indirect lighting techniques have also been implemented using rasterization hardware [EAMJ05, DS06] these approaches are quite limited. They only support a single bounce of light and completely ignore visibility after the first bounce because of the difficulty to compute it efficiently.

In summary, ray tracing is ideally suited to compute the visibility that is at the core of most rendering algorithms. Its close match to real world light transport and its physically-based computation allows for almost arbitrary flexibility when combining geometric models and shaders in a Plug'n'Play manner. Finally, ray tracing supports even extremely complex models efficiently and can be used efficiently for obtaining the indirect illumination that is likely to play an important role in future games due to its significant role in achieving higher realism.

3.3 Games using Ray Tracing Projects

In order to explore the requirements and benefits of ray tracing based games a number of experiments were performed. On the one hand several existing rasterization based games were reimplemented, including *Quake 3: Arena* [idS99], Quake 4 [idS06], and *Grand Theft Auto: Vice City* [Roc02]. On the other hand the game *Oasen* that assumes fast ray tracing hardware to be available and explores this new design space were developed.

3.3.1 QuakeRT

In 2004, a student project [SDP$^+$04] was realized which implemented a complete graphics engine using the OpenRT API in order to play a simplified version of Quake 3.

The goal was to fully support all graphical effects of Quake 3 as well as a collision detection system using ray tracing. Other non-graphic related algorithms like AI were simplified. Figure 3.5 shows some screenshots rendered on a PC cluster at 20 fps with a screen size of 640×480 pixels and $4\times$ full screen anti-aliasing.

Some features of this engine are realistic glass with reflection and refraction, correct mirrors, per-pixel shadows, colored lights, fogging, and Bézier patches with high tessellation. All of these effects are simple to implement with rudimentary ray tracing techniques. Additionally, two different light sources are supported: point lights for ordinary lighting and spot lights to simulate a flashlight. Other effects which can be defined with the Quake 3 shader language are also supported. Rather then using the original scene geometry some of the walls and floors have been replaced by highly tessellated geometry using static displacement mapping yielding approximately one million triangles e.g. for the *Temple of Retribution* map. The game engine was written from scratch and supports player and bot movement in-

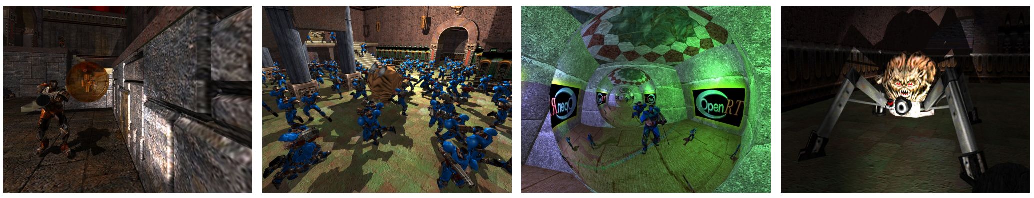

Figure 3.5: Various screenshots of QuakeRT featuring several shading effects like complex colored shadows from many objects, multiple reflective spheres, and spot lights.

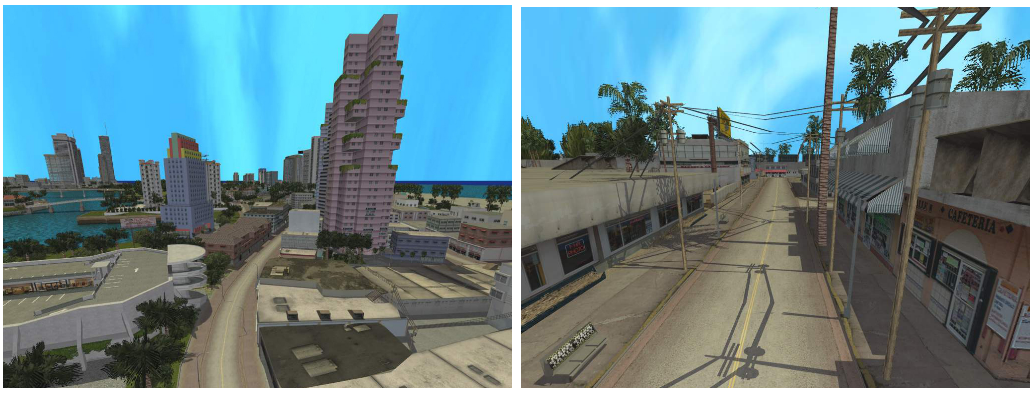

Figure 3.6: Example screenshots of Ray City.

cluding shooting and jumping as well as many special effects like jump-pads and teleporters.

The complete rendering engine was implemented by a single student within five months. Furthermore, the code complexity to support all these effects is very low. All effects were implemented independently and work together in a simple Plug'n'Play fashion.

Recently also *Quake 4* has been ported to work with OpenRT. In contrast to Quake 3, ray tracing was in this project not only used for the purpose of rendering, but also to use OpenRT's *ray shooting* capabilities for collision detection computation.

Using the same graphics engine, another project was started to reimplement GTA Vice City (see Figure 3.6). Special extensions to the graphics engine for this game are a normal mapped glass, water shaders, as well as a varnish shader for the cars. In contrast to the indoor Quake engine the Ray City implementation requires rendering large and complex outdoor environments under changing day and night illumination.

3.3.2 Oasen

Oasen [SDP$^+$04] is maybe the first game designed from scratch with the idea in mind to use realtime ray tracing as the core rendering engine. The player takes the role of a salesman on a flying carpet visiting different places, buying and selling goods while fighting off other players or bots. The landscape consists of several islands with trees, bushes, and buildings with more than 25 million triangles. No LOD techniques were used either for geometry or for lighting.

Main features of this game are realistic atmospheric simulation, day and night simulations including procedural stars and fireplaces at night, a depth-dependent water shader, procedural clouds, a living environment like swim-

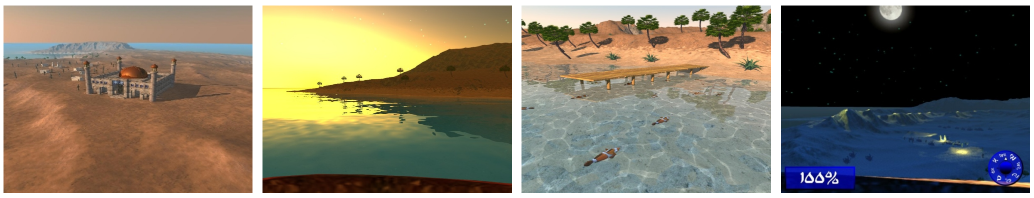

Figure 3.7: Some example screenshots from the ray traced game Oasen featuring several shading effects. These images are rendered on a PC cluster with 640 × 480 resolution at interactive frame rates.

ming fish in the ocean, and adaptive super sampling for anti-aliasing. Figure 3.7 shows some of these features.

The up to several hundred fireplaces during the night phase act of course as light source and are implemented as point lights. This huge number of light sources is efficiently handled by exploiting a restricted range of illumination for each light and organizing them in a spatial index structure. For each location it is then possible to efficiently find the light sources that contribute to its illumination.

This game was implemented within three to four month by four students during their spare time based on the experience with the Quake 3 project using the OpenRT API.

These experiments with ray tracing based games – while very preliminary – show that even scenes with advanced visual effects are indeed simple to create. Very little effort needs to be invested into selecting the correct algorithm to avoid visual artifacts. Even highly complex geometries and appearances can simply be modeled separately and work together without any additional effort.

3.4 OpenRT API

Today computer games communicate with the graphics subsystem on a higher level of abstraction through *application programming interfaces* (APIs) – most notably DirectX [Mic06] and OpenGL [WNDS01]. But although such interfaces feature powerful and flexible means for realtime 3D image generation, their design is heavily based on triangle rasterization algorithms. Unfortunately, this makes it difficult to use them also for interactive ray tracing due to fundamental algorithmic differences between ray tracing and rasterization.

An example of a graphics programming interface specifically designed for realtime ray tracing is OpenRT [DWBS03]. As the name suggests OpenRT is syntactically similar to OpenGL, but while staying as close a possible, it is neither a simple extension nor a subset of OpenGL. The OpenRT API itself is composed of two subinterfaces: the application interface and the OpenRTS interface for shading (see below).

The core *OpenRT application programming interface* assists an application in specifying geometric objects, textures, transformations, etc. in much the same way as OpenGL. In many cases the usual *gl* prefix can simply be replaced with *rt*. For example, most of OpenGL's geometric primitives are available, and are issued by implementing the same `rtBegin()` / `rtVertex()` / `rtNormal()` / `rtEnd()` statements. Despite these similarities in syntax, there are a few major differences in semantics.

3.4.1 Rendering Semantics

Ray tracing is fundamentally different because it uses global information, e.g. for shadows or indirect illumination calculations. Because of that, OpenRT's ray tracing back-end requires a more object-oriented approach. The user specifies a geometric *object*, which encapsulate primitives organized in a spatial index that can be seen as a low-level scene graph. Light transport simulation can be explicitly defined using programmable, dynamically linked shader plugins. Geometric objects are then bound to shader objects that contain specific attributes, e.g. material colors. This comes close to the *retained mode* of OpenGL where primitives are stored in *display lists* in a compiled form. The binding of individual shader instances may be regarded as *local* state changes, but two issues have to be kept in mind: First of all, OpenGL display lists also depend on global state and thus they can be rendered differently even if remaining unaltered themselves.

3.4.2 Objects and Instantiation

In general, a ray tracer cannot offer immediate mode rendering because all geometry needs to be defined before the actual light transport simulation can start. However, for scenes with a *manageable* triangle count this constraint has been somewhat relaxed lately due to new fast index structure build algorithms like [PGSS06, GPSS07, IWP07] – at least to a certain extend. OpenRT's object definition scheme behaves mostly like OpenGL's display list handling – except that there are no side effects due to global state changes. Primitives are grouped into objects, providing a collection of geometry plus associated shaders. Once an object has been fully defined, an acceleration structure is built, which is necessary for efficient ray surface intersection calculations. In the past this was a major drawback because if the object changed, its acceleration structure had to be rebuild completely from scratch (see Section 3.5).

Similar to calling OpenGL display lists, objects have to be *instantiated* in order to be effective. However, visibility computations will only take place

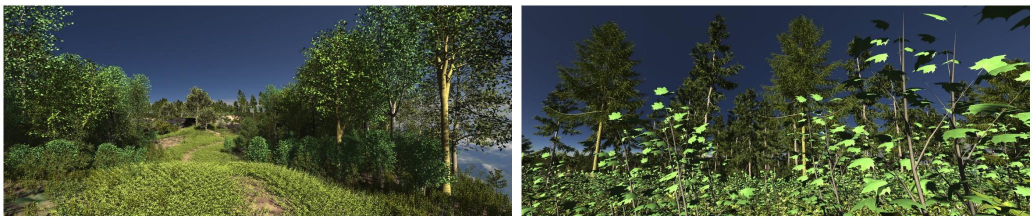

Figure 3.8: Forest scene with 365,000 plants and a total number of 1.5 billion triangles. Note the detailed shadows on leaves and the smooth illumination on the trunks due to integrating illumination from the entire sky.

if all objects have been specified. Instantiation works most efficient for a ray tracer (see Figure 3.8). Not only that a single object can be reused multiple times, because of inherent occlusion culling also no overdraw operations take place. Even thousands of instances can be used without suffering a major hit in rendering performance.

3.4.3 Multi-Pass Rendering vs. Shaders

The *OpenRTS shading language API* provides an interface between shaders and the ray tracing back-end to provide access to geometric and lighting information.

OpenRT provides a fully programmable shading model, making it possible to directly implement shaders for most optical effects. Writing such a shader is straight forward, and much easier than hand tuning complicated OpenGL code which often requires multiple passes for e.g. reflections. With ray tracing the multi-pass complexity can be put into local shaders that query for global data *as needed*. For example, adding reflection is one of the basic tasks of ray tracing. It only requires the shader to shoot on additional reflected ray, and can be specified by just a few lines of code. Listing 3.1 shows this in more detail.

One of the biggest advantages of ray tracing is the fact that independently written OpenRT shaders may also be simultaneously assigned to individual geometric objects in a simple Plug'n'Play manner. The desired effects are automatically combined and simulated in the correct order during ray tracing (see Figure 3.2).

3.4.4 Fragment and 2D Operations

Up to now OpenRT serves as a pure 3D graphics library. Games usually also apply 2D imaging *fragment* operations like stencil tests, alpha tests, or blending, etc. to perform special effects, e.g., alpha-blended explosions. Fragments, i.e. the output of OpenGL's actual rasterization stage can be regarded as partial color results in analogy to radiance values that travel along single rays, and therefore contribute to a pixel's final color.

```
class SimpleShader : public RTShader {
  //shader parameters
  R3 diffuse; // diffuse color (float r,g,b)

  RTvoid Register()
  {
    //export shader parameters
     rtsDeclareParameter("diffuse",PER_SHADER,
       memberoffset(diffuse),
       sizeof(diffuse));
  }

  RTvoid Shader(RTState *incident_ray)
  {
    R3 color,normal;
    R3 light(0.577f,0.577f,0.577f); //light direction

    //calculate diffuse term
    rtsFindShadingNormal(incident_ray,normal);
    color = diffuse * MAX(Dot(light,normal),0.0f);

    //add reflective term
    RTState reflection_ray;
    rtsReflectionRay(incident_ray,&reflection_ray,normal);

    color += rtsTrace(&reflection_ray);

    rtsReturnColor(color);
  }
};
```

Listing 3.1: A simple OpenRT surface shader example. The `Register()` function declares a diffuse color vector as parameter. Every time a ray hits a surface with which the shader is affiliated, the `Shade()` function is called. Reflections can be computed by a simple recursive invocation of the ray tracer (`rtsTrace()`).

OpenRT does not offer a direct equivalent to these operations, but they can be performed by programmable shaders. For example, blending can be realized using transparently textured polygons. Note, that it is also possible to mix OpenGL and OpenRT rendering.

3.5 Dynamic Ray Tracing

The interaction of the player with the 3D world is a key factor to build an immersive experience. Therefore, the use of ray tracing for games strongly depends on its ability to support dynamic scenes and animations.

Until recently this was considered to be a hard problem [SSM+05]. Some approaches already existed a few years ago [RSH00, LAM01, WBS03] but they are limited to a small subset of animation types (linear transformations only) or they are not fast enough to deliver interactive rendering performance.

The difficulty in dynamic ray tracing lies in maintaining spatial index structures that organize all scene geometry in order to significantly reduce the number of expensive ray intersection calculations. With their help typically only a few primitives need to be tested for intersection with a given ray even if the scene contains billions of them.

kD-trees are widely used for achieving realtime ray tracing performance for static scenes. However, their spatial subdivision approach made it difficult and costly to update it after changes to geometry.

By exploiting coherence with packets of many rays, competitive realtime ray tracing performance was also demonstrated with grids [WIK+06] as well as with BVHs [WBS07] that can also support dynamic scenes by exploiting fast index structure update strategies.

Additionally, two methods were proposed that allow for very fast updates to kD-trees. Firstly, a hybrid b-kD-tree acceleration structure combines the fast update properties of the BVH with the high ray tracing performance of the kD-tree and is already implemented in hardware [WMS06]. And secondly, *fuzzy* kD-trees together with a motion decomposition approach avoid the costly rebuilt of kD-trees almost completely at least for some types of animation [GFW+06] and skinned meshes [GFSS06]. These fuzzy kD-tree based approaches will also be discussed in detail in Chapter 4.

All these novel techniques provide realtime ray tracing performance for much more general dynamics than was ever possible before. The rendering of predefined animations, skinned meshes, and even more flexible deformations of the scene geometry are now supported by ray tracing. Thus ray tracing is at least competitive with rasterization in rendering dynamic worlds created by today's games.

Towards Fully Dynamic Game Environments

Traditionally games rely on precomputations to gain speed, e.g. they use precomputed radiosity solutions [CW93] stored in light maps or precomputed radiance transfer (PRT) [KSS02, SKS02, SLS05]. Precomputation is also necessary to allow for visibility culling, reducing the number of polygons sent to rasterization hardware by e.g. potential visible sets (PVS) [ARFPB90], binary space partition (BSP)-trees [FKN80], or portals [LG95].

Many games are in essence a walkthrough application, like first or third person shooters, with moving objects: Really changing the geometry such as destroying (parts of) buildings or digging holes into the landscape is seldom possible. Known exceptions are Red Faction with the GeoMod engine [THQ01] which allows for dynamic changes of the world, e.g. blasting holes in walls and Sega Rally [Seg07] where the racing tracks are not static but deform/deteriorate as cars are driving over them.

Because ray tracing computes visibility and simulates lighting on the fly the precomputed data structures needed for rasterization are unnecessary. Thus dynamic ray tracing would most likely allow for simulation-based games with *fully* dynamic environments as sketched above, leading to a new level of immersion and game experience.

3.6 Hardware Support for Ray Tracing

Based on the presentations in the previous sections it can be concluded that ray tracing is able to play an important role for future games. However, the most important question that still needs to be answered is that of suitable hardware support for achieving acceptable performance levels.

As a rough estimate for the following discussion it can be assumed that at least 300 million ray per second (1024×1024 pixels, 30 fps, 10 rays per pixel) are required to achieve a minimally necessary performance level. This is comparable to the performance of a low end graphics card for scenes with significant shadow and reflection computations. Based on experience, high-quality anti-aliasing may increase this number by about a factor of two if using an adaptive super-sampling strategy equivalent to 8 samples per pixel.

In the following it will be explored how different hardware approaches may be able to provide adequate computational power. Based on the performance data given in Section 3 it is in the following a currently achievable ray tracing performance of about 10 million rays per second on a single CPU/GPU for static scenes – and half of that for dynamic scenes – assumed. In other words an improvement in software performance by a factor of 30 to 60, respectively, or 60 to 120 with anti-aliasing is needed.

3.6.1 Multicore Architectures

For a long time CPUs have mainly tried to increase single-thread performance by driving clock rates up. However, this required longer pipelines, much larger caches, and complex technologies such as large reorder buffers, advanced branch prediction, and others to prevent problems due to the increasing cost of pipeline stalls. Also, these techniques all occupy precious die area that cannot be used for the main purpose of computing.

In contrast, many applications from computer graphics and other disciplines operate in a highly data parallel fashion. They are able to distribute their computations over an almost arbitrary number of threads as long as access to the necessary data can be provided. Ray tracing is a prime example of such an *embarrassingly parallel* algorithm as every pixel could be rendered independently from all others.

Due to that, a very interesting trend for ray tracing is that multi-core chips with four and more CPUs are becoming available. The roadmaps of the manufactures promise even more cores per chip in near future. In addition commodity multi-processor systems can be built from these, due to fast and low-latency CPU interconnects such as Hypertransport. Except for memory bandwidth and capacity, these systems will behave essentially identical to future multi-core systems when it comes to ray tracing. These systems can be used to get a rough estimate of the performance of future multi-core systems. For example, Dietrich et al. showed that 16 processors (eight dual-core Opterons) already allow for realtime ray tracing of a complex ecosystem with more then 1.5 billion triangles [DCDS05] (see Figure 3.8).

Given these promising hardware developments this year's dual-processor, eight-core system (16 CPUs) will probably be able to achieve the same performance and be able to match the minimum performance without anti-aliasing. Of course, such a system would be beyond the game market but it shows that even pure software ray tracing on standard processors is able to achieve reasonable performance levels for gaming in a short time frame.

A different multi-core architecture has been developed by IBM in conjunction with Sony and Toshiba: The Cell processor. It is an asymmetric design with a less-powerful management processor and eight high performance Synergistic Processor Elements (SPE) that offer SIMD instructions, operate solely on a limited local store of only 256 KB, and use fast asynchronous DMA for memory transfers and communication (up to 20 GB/s) [KDH+05b].

During Siggraph 2005 the German company inTrace [inT06] already showed a very early prototype implementation of a ray tracer running on the Cell. Even though this implementation was not fully optimized and supported only primary rays, it already achieved promising performance at full screen resolution for the Conference scene with a very simple shading (see Figure 3.1). In Chapter 5 a new approach based on the BVH will be discussed in detail.

3.6.2 Ray Tracing on GPUs

Graphics processors (GPUs) have evolved to highly parallel, programmable high-performance processors that can also be used to execute non-rasterization tasks. The latest GPUs contain 240 parallel processors that work together in a SIMT (Single Instruction Multiple Threads) fashion and offer an enormous raw compute performance. Purcell showed that ray tracing and Photon Mapping can indeed be implemented on GPUs [PBMH02, Pur04]. However, even though several other attempts have been made, e.g. [FS05], in recent years the ray tracing performance has stayed significantly below that available from current CPU implementations. However, the programming model and GPU architecture have changed dramatically. Exploiting new programming paradigms like CUDA [Buc07], that offer a direct way to program GPUs, ray tracing performance was able to speed up significantly. The performance is now at least on par with (single-core) CPUs [PGSS07, GPSS07] and sometimes even faster. NVIDIA showed at Siggraph 2008 a highly optimized GPU ray tracing demo that achieved realtime framerates even for a complex scene with more than 2 million triangles and complex shading. Today, a GPU is not a special purpose rasterization hardware anymore that can also execute other algorithms, but a highly-parallel multi-core processor that can also perform rasterization using some specialized blocks of hardware.

3.6.3 Custom Hardware for Ray Tracing

Currently, only a single company [ART03] markets dedicated hardware solutions for ray tracing. However, this product is designed for accelerating offline rendering and is not suitable for realtime interactive applications.

An entire family of custom realtime ray tracing processors has been designed by Schmittler and Woop at al., ranging from a non-programmable architecture [SWS02] and its FPGA realization [SWW$^+$04] to a fully programmable architecture (DRPU) [WSS05] and its extension that also supports fully dynamic scenes [WMS06]. All these designs have been realized using FPGA technology and achieve realtime performance (see Figure 3.9).

The hardware architecture of the fully programmable RPU design includes on-chip ray generation, kD-tree traversal, programmable intersection computations, arbitrary levels of reflection or transparency, and fully programmable shading. The RPU is designed essentially as a highly parallel processor that efficiently supports recursive ray tracing computations. The hardware can exploit the usually large coherence in ray tracing by a SIMD technique that adaptively splits packets as they may become incoherent during the computation. This programmability of the RPU supports also programmable geometry and image processing.

The extended RPU design [WMS06] also efficiently supports dynamic scenes due to its use of b-kD-trees (see Section 3.5). The update of changed geometry as well as general skinning operations are directly performed in hardware. The update is extremely cheap such that even on an FPGA mil-

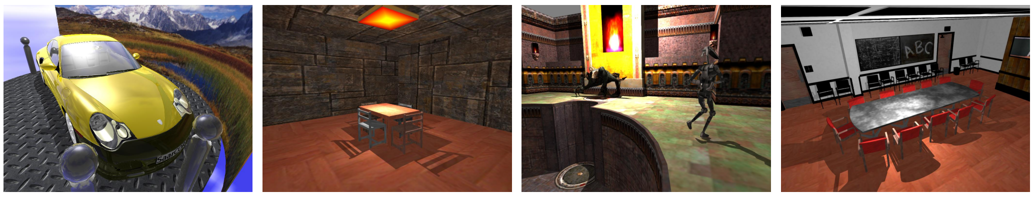

Figure 3.9: Four example images rendered with the RPU design at interactive frame rates at a resolution of 1024 × 768.

lions of triangles can undergo transformations without introducing a bottleneck.

The performance of the FPGA prototype running at only 66 MHz compares well with that of OpenRT running on a processor with 40x the clock rate (see Table 3.2). Current research investigates the performance expected when implementing the RPU architecture in ASIC technology. Preliminary results show that a completely unoptimized ASIC with twelve RPU engines should already be able to achieve between 62 and 250 million rays per second at a clock rate of 266 MHz and at a resolution of 1024 × 1024 pixels.

Note that these estimates are for an initial prototype of the RPU architecture in an ASIC process without optimized compute-blocks like floating-point adders or multipliers. There is still a significant gap between these performance numbers and those from current GPUs. However, they are very encouraging given that these numbers from first prototype are compared to chips that have been refined over many years.

The estimated performance numbers above are interesting in a number of ways. A pure software solution is still greatly limited by the number of processing cores available in a system. It seems that more aggressive multi-core designs are required in order to overcome this bottleneck. The Cell processor with its eight cores is promising but is still not fast enough for gaming purpose (see Chapter 5). In the non-gaming environment larger multi-processor machines should be able to achieve the necessary performance levels for fully

Scene	Scene6	Office	Quake3	UT2003	Conference
Triangles	0.8k	34k	39k	42k	282k
FPS	5.2	3.6	3.1	1.8	1.3

Table 3.2: Performance results measured in million rays per second for the FPGA prototype of the RPU clocked at 66 MHz at a resolution of 1024 × 1024 pixels. Note, that the performance is already comparable to that of OpenRT even though that runs on CPUs with 40× the clock rate.

interactive simulations. However, it is clear that these systems will be too expensive in order to be interesting for a mass market. The situation is somewhat different for dedicated ray tracing hardware, though. It can be assumed that these performance numbers for dedicated ray tracing hardware have a very large potential to increase significantly. More efficient hardware implementations and optimizations to the architecture are two obvious strategies.

All of these approaches are likely to benefit greatly from future approaches to further minimize the number of rays that need to be computed for a specific effect and further optimizations of the core ray tracing operations. It can be assumed that we are just starting to see the potential of realtime ray tracing – in games and other applications.

3.7 Conclusions

In this chapter it is argued that the dramatic increase in ray tracing performance seen over the last few years together with current trends in hardware development should make it interesting to take a new look at the use of ray tracing for gaming applications.

This argument is supported by analyzing the benefits of ray tracing compared to current technology and by presenting key developments that are important in the context of game development, including application interface issues and support for dynamic scenes. Additionally, experience from several prototype game projects were presented that support the conclusion that the physics-based approach of ray tracing greatly simplifies the development of game engines as well as the design of game content.

This Plug'n'Play approach for content design together with the ability to efficiently and accurately simulate even advanced effects and complex lighting situations without the danger of introducing objectionable artifacts is a major bonus for simulation-based games, which increasingly avoid precomputation and instead rely on runtime simulation of the effects.

Unfortunately, the current performance level of ray tracing is not yet sufficient to sustain a larger market for gaming. However, the current trends towards high-performance parallel hardware strongly support and complement the improvements on the algorithmic and implementation side. However, given the lead times in game and hardware development, it seems that the time may be right to explore ray tracing based games in a wider context.

Finally, it is clear that realtime ray tracing is still an emerging field where major breakthroughs are still likely. It is also an area that still requires significant future research, including further improvements in the handling of dynamic scenes, even faster simulation techniques for global and indirect lighting effects, efficient anti-aliasing that decouples visibility samples from the shading samples, and of course further performance improvements for all parts of the ray tracing pipeline.

Chapter 4

Ray Tracing of Dynamic Scenes

In Chapter 3 it is pointed out that the ability to render dynamic scenes efficiently is a requirement for a serious use of ray tracing in computer games. This chapter describes a novel approach that preprocesses dynamic scenes such that, while rendering, only very little on-the-fly computations are necessary. The preprocessing is in particular designed for animation sequences and skinned animations as they are typically used for computer games.

4.1 Spatial Index Structure Considerations

As seen in the Chapter 2, ray tracing requires (hierarchical) spatial index structures to achieve realtime rendering performance. Due to the nature of dynamic scenes to change over time, an index structure built for a particular point in time may not be valid for the next one, as the location of the geometry changed. This implies: for each new frame a new index structure must be build. Unfortunately, the construction time complexity – for the hitherto most efficient hierarchical index structures – is $\mathcal{O}(n \log(n))$ for both kD-trees [WH06] and BVHs [Wal07]. This renders a complete rebuild, at least for non-trivial scenes, impractical. Grids can be constructed in $\mathcal{O}(n)$ [WIK$^+$06] but are in average somewhat less efficient in terms of rendering performance, and suffer from the "teapot in a stadium" problem since they do not adapt to locally different geometric densities and object sizes in a scene. Recently it was found that a BVH, once built, can simply be updated to support efficiently certain kinds of dynamic scenes [WBS07]. Nevertheless, all this approaches depend on the number of primitives a dynamic object consists of. In contrast to these approaches the below described *motion decomposition* approach supports dynamic scenes for **all** kinds of hierarchical index structures that can be build over axis-aligned bounding boxes and is – while rendering – *independent of the number of primitives* and thus avoids the need of a **complete** rebuild or update.

4.2 *Dynamics* Classification

In order to render dynamic scenes efficiently it is advantageous to have some knowledge about the dynamic content. If nothing is known, one can only treat the dynamics as black-box and rebuild the index structures from scratch. However if the dynamics is known, either because it is completely predefined or at least some algorithmic constraints are known, it is possible to derive some information from the animation and it has to be decided *when* and *how* this information is utilized. Naturally, this can be done in a preprocessing phase or while rendering.

The dynamic scenes that are considered in the context of this chapter – namely 3D computer games – consist of individual objects, each consisting of a number of connected triangles, that change its position with time. Nevertheless, it is not allowed that the number of triangles or its connectivity changes throughout the animation. An example of such an object can be seen in Figure 4.1, a human performing a running motion.

In the following *dynamics* is classified by the kind of deformations that are applied to the triangle meshes, or how the motion is specified for processing. Furthermore, it is described what kind of solutions exist to ray trace these kinds of dynamic scenes:

No Deformations: The simplest deformation for a scene is simply no deformation. Static scenes can be efficiently ray traced today using data structures like kD-trees [RSH05, Res06] or BVHs [WBS07] and their related coherent algorithms as discussed in Section 2.4.3.

Affine Transformations: In computer games it is very common to apply affine transformations to objects e.g. to translate, rotate, and scale them on-the-fly based on some physical models. Lext et al. [LAM01] proposed a two-level approach, using a hierarchy of oriented bounding boxes (OOB), where each individual object gets its own acceleration structure and a final top-level structure is build over all the individual OOBs. This allows, for example, to support hierarchical robot like animations of avatars in computer games. The top-level construction complexity is then only dependent on the number of objects in the scene and not on the number of individual primitives, e.g. triangles a scene consists of. If an affine transformation is applied to an object, rather than transforming all its primitives, only its bounding box has be transformed. After transforming all OOBs a new top-level structure can be build. A ray that traverses the top-level structure and reaches a top-level leaf node is then transformed to the local coordinate system of the object whose associated index structure has to be traversed next. Wald et al. [WBS03] used a similar two-level approach with kD-trees and achieved for this kind of animation for the first time interactive frame rates on a desktop PC.

Random Motion: Another form of motion can be found in particle and turbulence flow systems. In contrast to typical CAD or 3D computer games scenes, these scenes consist of very many individual particles whose motion is steered by some mathematical simulation e.g. in flow field visualizations. Moreover new particles may be generated on-the-fly or disappear such that the number of particles in the system depends on the state of the simulation. Reinhard et al. [RSH00] proposed the *interactive grids* approach to handle such particle systems efficiently. A special feature of this approach is that the grid – once build over the particles in an initial state – can *grow* with particles that would be moved (or generated) outside the initial grid in the simulation. To insert a particle, outside the initial grid, a wrapped position in the initial grid is computed. A then newly defined *virtual* grid over the new real bounds is then used for ray traversal. In order to achieve interactive frame rates Reinhard used an SGI Origin 2000 with 32 processors. Today, exploiting a coherent grid traversal [IKW07] would probably achieve the same, or even better, performance on a single modern PC. If no information is present to exploit *tricks* – like transforming a ray instead of thousand of triangles as described above – the grid seems to be the best choice for visualizing particle simulations and related problems.

Fixed Animation Sequences: For computer games it is quite common to specify an animation as a fixed set of animation steps – the keyframes – e.g. for a walking-sequence of an avatar. No further information, except the object description for each time step is available to rendering system. Until recently, the two best possibilities to ray trace this kind of animation were to: build an own acceleration structure for each time step of the animated object and choose the right one dependent on the timestamp while rendering, or to build a new acceleration structure on-the-fly as it is needed. Both approaches are not optimal – at least for computer games that require realtime performance with additional restrictions to the available host memory: Either too much memory for all the preprocessed index structures is required, or the overall rendering performance is reduced due to a separate build for each animated object before the rendering can take place. Properties that were not exploited until recently are: that these animation sets are known in advance, have always the same number of triangles, share the triangles connectivity across the animation, and the animation itself comprises a certain kind of *locality*. Locality means in this context that triangles that are locally close together (largely) follow the same, or at least a similar trajectory. Until recently no algorithm was known, that exploits these properties to ray trace

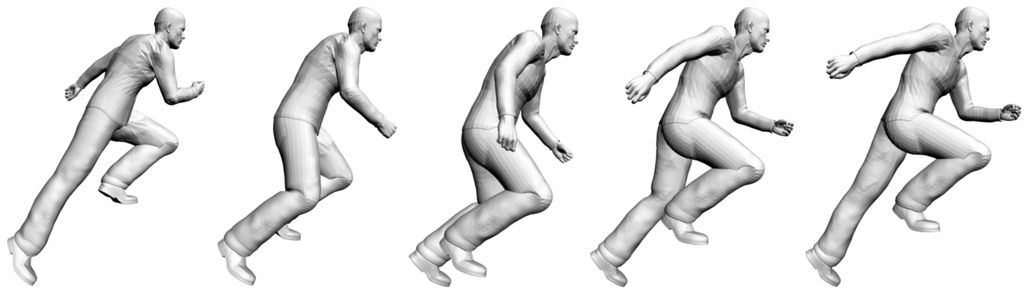

Figure 4.1: A typical (simple) example animation as it can be found in a computer game or animated movie. The animation is given as a set of individual triangle meshes – the keyframes. Each keyframe has its own set of triangles, but throughout the animation the connectivity between the triangles is fixed.

keyframed animations.

Skinned Animations: Mesh animations that are based on an underlying skeleton are also frequently used in computer games. Examples are not limited to avatars but also plants like trees, grass, and many other things that can be animated by fitting a skeleton to its *limbs*. In a first step, a designer generates a model of the object that is to be animated. Then, a skeleton, consisting of *bones*, is build and fit to the model. Generally, but not necessarily, this skeleton builds a single hierarchy like a human skeleton.

To make the motion of the vertices dependent on the bones, a set of vertices is assigned to each bone in the fitting step. A single vertex can belong to multiple bones, e.g. vertices that are close to the joint of two bones. To control the motion of the vertices, each vertex has an associated *weight* per bone that controls its influence. A modeler can now design different animation sequences with just a few keyframes per sequence, and new interpolated poses can just be computed on-the-fly without manual intervening. If a transformation is applied to a bone, all associated vertices follow this motion based on the associated weights (see Section 4.5 for more details).

The advantage of this skinned animations is that they allow to interpolate new poses between two keyframes that are not explicitly defined, and to blend seamless different motion sequences like the transition of a walk sequence to a jump of an avatar. Similar to static animation sequences, no algorithm was until now published that targets explicitly at ray tracing skinned animations using its inherent properties.

4.3 The Motion Decomposition Approach

In the last section, two classes of dynamic scenes have been identified, animation sequences and skinned animations, that are important for computer games. Sections 4.4 (animation sequences) and 4.5 (skinned animations) will present a solution to ray trace these kind of animations exploiting their inherent properties.

4.3.1 The Idea

In contrast to other dynamic ray tracing methods that were recently published, e.g. [HMS06, LYTM06, YCM07, ISP07, IWP07] to name just a few, the motion decomposition is a *preprocess*. The goal of this preprocess is to avoid the dependency between the number of transformed primitives in the scene and the acceleration structure rebuild. At the core of the motion decomposition approach is the idea to exploit the local coherence that can be found in the dynamics of many animation sequences and skinned animations. Consider the simple animation in Figure 4.2 that consists of two keyframes. At first the triangle mesh is in its initial position. Then a transformation T is applied to the mesh. Basically the motion of the mesh follows a simple affine transformation (rotation and translation) plus some additional motion for two vertices of the orange triangle. In essence that means, this motion can be separated into two independent parts: a common affine motion and a remainder, or residual motion. Let's suppose an animation consists of two time steps S_1 and S_2. The idea is now to find a good mapping, i.e. an affine transformation A, that transforms S_1 to S_2. Using the affine transformation A solely will in most cases not result in a correct mapping because the real transformation from S_1 to S_2 was probably more complex, e.g. as in the example above. However, a resulting error can be catched by some Δ such that:

$$S_2 = \mathrm{A}S_1 + \Delta \tag{4.1}$$

can be used to transform S_1 to S_2[1].

In doing so, the motion of the mesh in Figure 4.2 could be described by the motion that can be covered by the transformation A plus a correcting term for the residual motion of the two vertices of the orange triangle that is captured in Δ. Of course this method can also be applied if the animation consists of n time steps i.e. :

$$S_i = \mathrm{A_i}S_1 + \Delta_i \tag{4.2}$$

Then every time step simply requires its own transformation matrix $\mathrm{A_i}$ and Δ_i with $i \in \{1..n\}$.

[1]Please recall that $\mathrm{A}S_1$ implies the multiplication of A with every single vertex S consists of. Similarly, for every vertex in S, there is a corresponding δ which is basically an offset vector. The set of all δ is referred to as Δ. This notation is just chosen for simplicity.

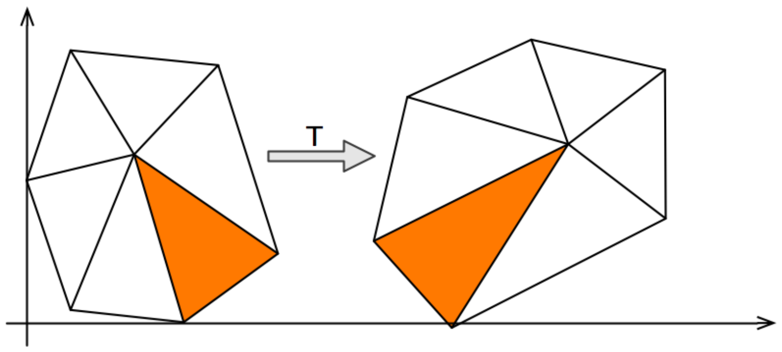

Figure 4.2: A simple animation sequence. The mesh on the left (mesh one) is transformed with T resulting in mesh two. Please note that T is not just affine. T includes a common rotation and translation for all vertices and an additional transformation that is only applied to two vertices of the orange triangle.

4.3.2 ***Fuzzy*** Bounds and Positions

What does that mean to ray tracing? Using Equation 4.2 the motion of an animation is decomposed into two parts. An affine transformation and some delta that corrects the error that a sole affine transformation would induce. In Section 4.2 it is described that dynamic objects can be ray traced using a two-level hierarchy of index structures, that use affine maps to transform bounding boxes and rays instead of triangles [LAM01, WBS03]. This approach is exactly what can be used for the affine transformations $\mathrm{A_i}$. The remaining problem is now what to do with Δ_i.

In Figure 4.3 it is shown how the Δ_i can be handled. On the left side, the previous animation example is shown. But this time each mesh has an associated bounding volume, e.g. a box that a leaf node in a BVH would comprise. As can be seen, the size of the volumes are different for both meshes. If mesh two is projected into the coordinate system of mesh one, it is possible to merge their bounding volumes such that a **single** box can be constructed that contains all the space that is required by the animation (Figure 4.3 right). More formally, by multiplying Equation 4.2 with $\mathrm{A_i}^{-1}$, S_i is transformed into the coordinate system of S_1:

$$\mathrm{A_i}^{-1}S_i = S_1 + \mathrm{A_i}^{-1}\Delta_i \qquad (4.3)$$

$\mathrm{A_i}^{-1}S_i$ is then the so called *fuzzy pose* of S_i, and the union of all bounding volumes results their *fuzzy bounds*. It is now possible to construct for every triangle in the scene its fuzzy box that includes all the space that is required for its animation. Now, rather than constructing an index structure over the

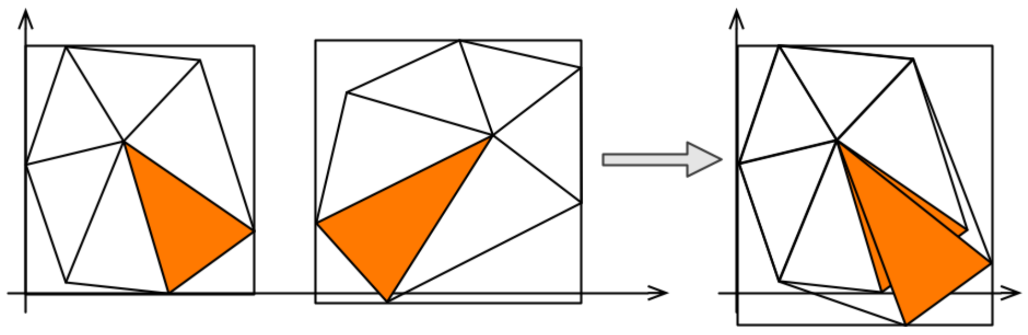

Figure 4.3: The simple mesh animation sequence including their bounding boxes (left). When time step two is transformed into the coordinate system of time step one, using $\mathrm{A}_\mathrm{i}^{-1}$, the union of the bounding boxes results in a new one that is valid for the whole animation (right).

scenes primitives, the index structure is build over **all** positions of $\mathrm{A_i}^{-1}S_i$ of a primitive. This results in an index structure that is valid for the complete animation.

4.3.3 Clustering

Unfortunately, most animations are much more complex, both in motion and size. Consider the hand motion sequence in Figure 4.4. This animation is given as a set of triangle meshes consisting of more than 15,000 triangles each. All fingers move in the animation at the same time to different positions. Additionally, each single finger comes with three joints (except the thumb with only two) that move differently. When each triangle is now treated as a separate object and gets its own fuzzy box this would result in huge fuzzy boxes with a lot of overlap. Even though a resulting index structure, that could be build over these huge fuzzy boxes, would be valid for the whole animation, the ray tracing performance would be poor since too many triangles would have to be tested for intersection due to the too large and heavily overlapping fuzzy boxes. This would, in a worst case scenario, come close to the case where no index structure is used at all for ray tracing.

What is needed is a way to minimize the size and overlap of the fuzzy boxes. An extreme would be to reduce the overlap to zero by avoiding the fuzzy boxes and just build the index structure over the hands triangles, or their tight bounding boxes. Unfortunately, this requires then again a full rebuild of the index structure for every frame since the index structure is then again only valid for one particular time step. In essence, this means that there is a overlap-rebuild tradeoff. The question is now if there is an optimal setting that minimizes the size and overlap of the fuzzy boxes while avoiding a full rebuild of the index structure.

When the motion of the hand is observed, it can be seen that there are sets of triangles that follow basically the same trajectory. Triangle areas that follow the same trajectory will probably have a very small Δ after compensating their common motion using an affine transformation. This yields the idea to identify those coherently moving mesh regions that produce a very small overlap in their fuzzy boxes. Having identified these regions, they can be treated as separate objects that will have for each time step a separate index structure, optimized bounding volume, associated transformation, and deltas.

Then, for each new frame a new top-level index structure is build over the current bounding volumes that the regions cover in the spirit of Lext and Wald [LAM01, WBS03].

In doing so, the overlap of the bounding volumes in the top-level structure is minimized. The index structures of the individual regions **have not** to be rebuild since they are constructed using the fuzzy bounds of the triangles. This is acceptable since the number of these coherent moving regions, and thus the number of bounding volumes that have to be used to construct the top-level structure is usually small (see Sections 4.4.4).

In the next two Sections, 4.4 and 4.5, details of the motion decomposition will be discussed with respect to animation sequences and skinned animations. In the following several key questions have to be answered: How can the transformation matrices $\mathrm{A_i}$ be computed, what is a good strategy for identifying regions of coherent motion in the animation, and is it really a good idea to compute all mappings with respect to the first keyframe or should another keyframe be used, which one?

4.4 Motion Decomposition of Animation Sequences

In this section, the answers to the above mentioned questions are given with respect to animation sequences. To make the motion decomposition work, there are constraints the animation has to meet. It is not allowed that the number of triangles or their connectivity changes across the animation. But most importantly, the more locality can be identified in the time animated mesh, the better the algorithm will work. If the animation behaves like a particle system – with many individual objects – that do not have coherent but random motion, the algorithm would not be as efficient, because the properties that should be exploited are not present. In the next four sections missing details about the motion decomposition and results are presented.

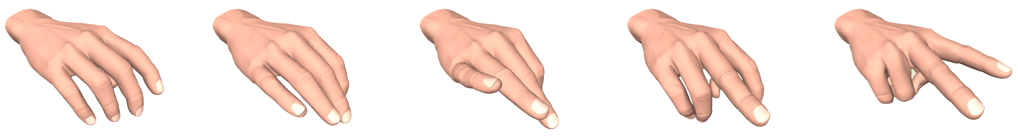

Figure 4.4: An example of an animated hand: Due to the complexity of the animation the complete mesh has to be segmented into regions of coherent motion such that the fuzzy bounds are small and their overlap is minimized.

4.4.1 Calculating $\mathrm{A_i}$

In Equation 4.2 a mesh S_i is transformed to a reference mesh S_1 by an affine transformation plus some additional motion Δ_i. But until now, it is not clear how the $\mathrm{A_i}$ can actually be computed. Since we deal with triangle meshes in 3D and want to have affine transformations, each $\mathrm{A_i}$ has to be a homogeneous 4×4 transformation matrix. Furthermore, as described above, the Δ_i should be as small as possible to avoid large overlapping fuzzy bounds. A criterion to obtain a *good A_i* is to minimize the **squared** sum of the length of all offsets vectors in each Δ_i. This leads then to a linear least squares problem (LLS) [Str03]. If Equation 4.2 is solved for Δ_i we can write the formula as:

$$\|\Delta_i\|^2 = \|\mathrm{A_i} S_1 - S_i\|^2 \tag{4.4}$$

Essentially, Equation 4.4 expresses Δ_i as an error that an affine only transformation would induce when transforming S_1 to S_i. This LLS optimization problem can be solved using conventional mathematical techniques like the *singular value decomposition*. The result of the LLS computation is a $\mathrm{A_i}$ with a minimal $\|\Delta_i\|$. This is achieved by minimizing the sum of the squares of the offsets, i.e. the differences $\mathrm{A_i} S_1 - S_i$. A big disadvantage of the LLS method is that it is in general prone to outliers, which results from squaring the errors. Outliers have then too much importance. Fortunately, here the $\mathrm{A_i}$ are not affected because outliers will be removed automatically as they will be identified in the mesh segmentation step and act as seeds for new clusters (see Section 4.4.2 for more details).

4.4.2 Mesh Segmentation

Mostly all interesting animation sequences are large and complex in its motion trajectories and size. As described before, complex meshes have to be segmented into coherently moving parts to achieve an optimal ray tracing performance. A commonly applied technique for mesh segmentation is *clustering*. Clustering is a fundamental data analysis tool used for many scientific problems [KNT05] and can be used to classify objects according to perceived similarities [JD88]. In the case of this work, the similarity is the coherent motion of connected triangles. It is possible to think of many ways to guide a clustering process for generating appropriate triangle clusters. Here, a cluster is a number of connected triangles that have a *similar* motion across the animation sequence and thus a small Δ. Now there are two options for the clustering. Either a fixed number of clusters could be generated, or new clusters are generated until a certain criterion is reached. Here, the second method is used. The goal of the clustering process is now to identify those regions in a mesh that minimize the sum of all Δ_i. Since the clustering process generates new submeshes the notation is now changed to ${}_k\Delta_i$ and ${}_k\mathrm{A_i}$. The k indicates then the number of clusters that are used for the object.

Among all the possible clustering methods the generalized Lloyd's algorithm [Llo82] is a suitable candidate to cluster the triangles into coherently moving submeshes. Lloyd's algorithm is an iterative clustering approach and has four parts: *initialization*, *partitioning*, *refitting*, and *repeat*.

Initialization: Since the optimal number of clusters is not known in advance, just a single cluster containing the fuzzy bounds of all triangles is generated to initialize the clustering process.

Partitioning: In the partitioning phase all ${}_k\mathrm{A_i}$ are computed and each triangle is assigned to the cluster where it has the smallest residual motion. This process is repeated until the sum of all ${}_k\Delta_i$ does not decrease significantly anymore or a (user defined) maximum number of iterations is reached. Please recall that the ${}_k\mathrm{A_i}$ are 4×4 matrices and that a unique solution requires at least four vertices, e.g. two triangles, in the LLS computation. Therefore, if a cluster has only one triangle, one of its neighboring triangles is also assigned to this cluster.

Refitting: After distributing the triangles optimaly to the existing clusters in the partitioning phase, a new cluster is now generated. The seed for the new cluster is the triangle that has the **largest** residual motion. For all the old clusters a new seed is chosen as well. A triangle in an old cluster will be chosen as new seed when it has the **smallest** residual motion. Then the triangles are distributed again to the clusters as in a partitioning step.

Repeat: Partitioning and refitting will be repeated until the clustering converged. In the current implementation the clustering is considered converged when the sum of all ${}_k\Delta_i$ does not decrease more then a user specified threshold i.e. 1%. Figure 4.5 shows an example run for the hand animation.

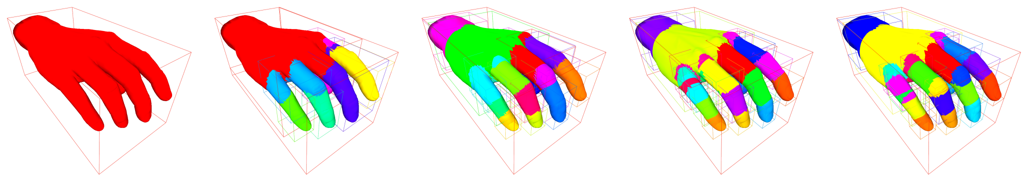

Figure 4.5: An example run of the clustering algorithm: As the number of clusters increases the surface area of the bounding volumes shrinks until a point where new clusters are not beneficial anymore. To depict the clustering process, each new cluster and bounding volume is colored differently.

4.4.3 Selecting the Reference Mesh

Until now it was alway said that a mesh S_i should be mapped to a mesh S_1 for computing the $\mathrm{A_i}$. If a set of keyframes is given, as in Figure 4.1, it is not very likely that the first keyframe is the optimal mesh in the set to which the other meshes should be mapped to. So, what is the reference frame S_x that is used best for the mapping computations?

A priori it cannot be known which keyframe will produce the best result in the clustering process. If preprocessing is not time critical, every cluster cycle is executed for each keyframe and the one that minimized the surface area of all clusters is chosen. This makes sense since in general the surface area of the bounding volumes and their overlap is strongly coupled. When the animation consists of too many frames such that the preprocessing time would not be acceptable the clustering uses a simple uniform sampling approach, that can be guided by the user, and only every js frame is tested. Of course it may be that the optimal reference frame is not found. But still a good result can be obtained because most animations have a *smooth* behavior such that close to optimal results are achieved.

4.4.4 Results

In order to test the efficiency of the described motion decomposition approach several test scenes from small to large, and simple to complex animations, have been chosen (see Figure 4.6). As index structure the kD-tree is chosen although BVHs could also be used. All algorithms – including the motion decomposition algorithm and ray tracing engine – are implemented in C++ using SIMD extension whenever useful. The ray tracing engine uses the ray-triangle intersection test of Wald et al. [Wal04], the surface area heuristic for building (boxed) kD-trees [MB90, Ben06], and the inverse frustum culling approach for ray traversal [RSH05] (see Chapter 2 for more details). For measurements a 2.8GHz Opteron PC is used, equipped with 2GB of RAM

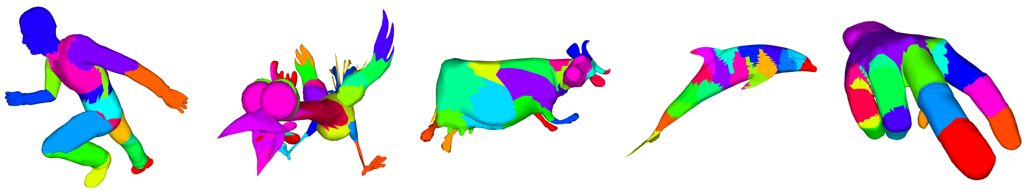

Figure 4.6: Example images of clustered test scenes. Each colored region represents a separate cluster. From left to right: Ben, Chicken, Cow, Dolphin, and Hand.

and a GeForce 6800GT graphics board that is solely used for displaying the rendered images. Next, the used test datasets are briefly described:

Ben, Hand: The Ben and Hand animations comprise both natural motion. Ben is a running sequence as it could be found in a typical computer game or animated movie. In the hand animation the fingers just bend and stretch back to its initial position. Both animations consists of 30 keyframes. The clustering procedure stopped for the Ben model with 20 and the Hand with 21 clusters. Ben consists of roughly 78k triangles and the Hand of 15k.

Chicken: The Chicken animation is a cartoon like animation with typical squeeze and stretch elements resulting in unnatural poses of the chicken. 400 keyframes are used for the animation and consists of several different motion sequences like walking, running, and fluttering. Each keyframe consists of approx. 56k triangles that are segmented into 21 clusters.

Cow, Dolphin: Both, Ben, Hand, and Chicken look like animations that are designed with a skeleton that is fitted to the models. The Cow and Dolphin animations are computed based on some physical simulations. Although the swim-animation of the dolphin still looks realistic the cow is almost arbitrary squeezed, stretched and skewed. The cow consists of 204 and the dolphin of 101 keyframes resulting in 20 and 16 clusters.

Clustering Efficiency

In Section 4.4.2 it is claimed that if the sum of all ${}_k\Delta_i$ is minimized, the surface area of the clusters shrinks and is therefor a good optimization criterion. Figure 4.7 shows this correlation.

As can be also seen in Figure 4.7, a number between 20 and 25 clusters is in general sufficient. This allows also a very quick rebuild of the top-level structure per frame. The running time of the clustering preprocess is highly dependent on the size of the meshes with respect to the numbers of triangles and keyframes. The Hand and Cow animations can be clustered in approx. 20 minutes. Nevertheless, the Chicken sequence requires up to 90 minutes which is still a reasonable timeframe.

As the images in Figure 4.6 show, the final clustering of the Hand, Ben, and Chicken models fit nicely to a skeleton structure. Although the Dolphin and Cow model do not show such a skeletal based clustering, the clusters are absolutely usable in term of ray tracing performance.

Overall Performance

Maybe the most important measure is the total ray tracing performance in FPS that can be achieved using the motion decomposition approach. Figure 4.8 shows the performance for all the test datasets across the whole animation. Easily it can be seen that the performance is always at least in an interactive range from six to 50 FPS, rendered with a screen resolution of 1024^2 pixels and simple diffuse shading.

The frame rate for the Ben, Hand, and Dolphin animations are more "stable" compared to the Chicken and Cow sequences that show larger fluctuations. In the Chicken animation the frame rate increases drastically in the last frames because the chicken runs away from the camera such that it only covers a small fraction of the screen area anymore. Similarly the large differences in the frame rate of the Cow animation can be explained since the number of covered pixels varies strongly with the animation.

Performance Comparison

Another interesting measure is how much the rendering performance is reduced when a fuzzy index structure is used in comparison to an optimized index structure for a single keyframe. For this measurements, a separate kD-tree is build for every keyframe and the performance measured in terms of traversal steps, intersected triangles, and finally the rendering performance in FPS. Table 4.1 shows this numbers averaged for the whole animation in comparison to the same numbers measured using the motion decomposition

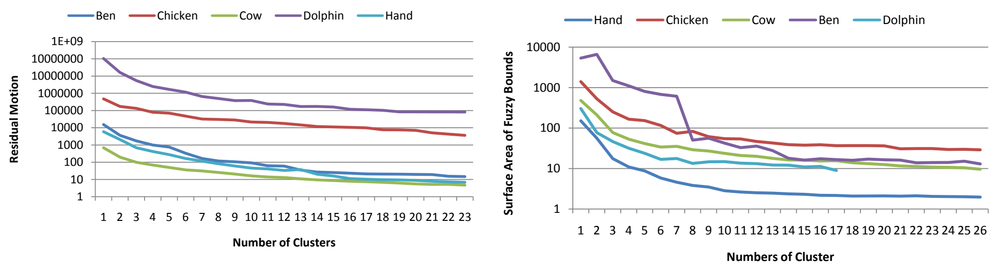

Figure 4.7: The correlation between the overall residual motion Δ and the sum of the surface area of the fuzzy boxes. As the residual motion decreases, the surface area shrinks in a similar fashion.

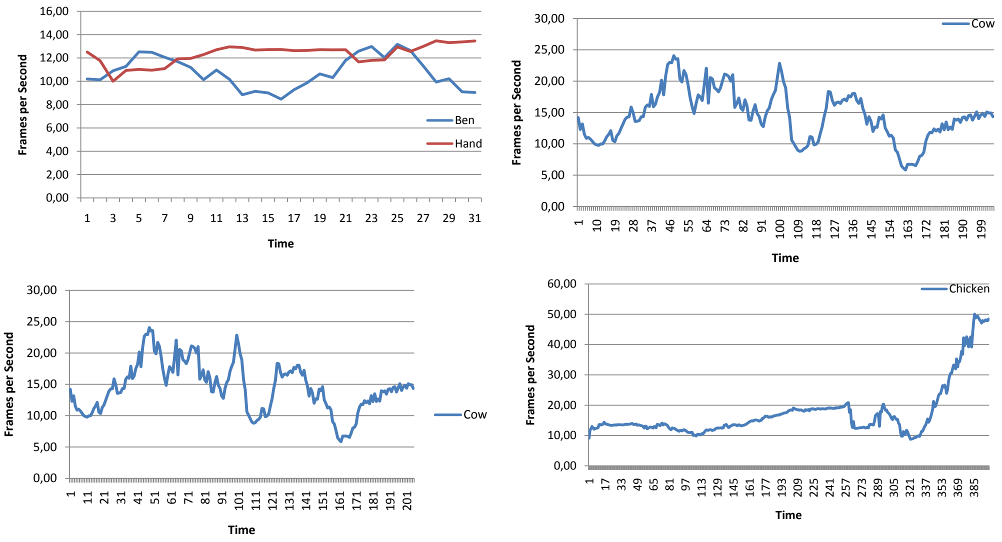

Figure 4.8: Overall rendering performance timelines for the test datasets. All datasets are rendered with a screen resolution of 1024^2 and for shading a simple diffuse model is used. As can be clearly seen, the ray tracing performance is always in a range from six to 50 fps.

approach.

As can be seen in Table 4.1, the ray tracing performance just drops down by a factor of 2.6 at most. This is an absolutely acceptable result considering that a full kD-tree rebuild typically would be more expensive. For shorter and less complex animations, all statistical results are in an expected range as the number of traversal steps and triangle intersections just double. The Chicken and Cow animations stress the motion decomposition approach due to their length and different animation sequences. In contrast the Ben, Hand, and Dolphin only consist of a single animation sequence e.g. the hand just opens and closes. Nevertheless, even the Chicken animation can be ray traced on a single CPU-core fluently since the increase in intersection tests does not hurt significantly.

4.5 Motion Decomposition of Skinned Animations

In the last section, a preprocess was discussed that allows to ray trace dynamic scenes efficiently that are completely predetermined by a set of keyframes. Now, skinned animations will be considered that also consist of keyframes but provide additionally more information and a certain freedom of the motion. Rather than relying solely on keyframes, skinned animations possess a skeleton that allows to extract information about the motion (see Figure 4.9 for an example). This allows two things: first, to interpolate be-

	#Traversal steps			#Intersections			Average FPS		
Scene	Static	Fuzzy	Ratio	Static	Fuzzy	Ratio	Static	Fuzzy	Ratio
Ben	722,068	1,111,808	1.54	1.2	2.28	1.9	20.98	10.77	1.94
Chicken	210,458	379,466	1.80	0.80	4.96	6.2	39.24	15.03	2.61
Cow	634,173	946,558	1.49	0.92	3.68	4.0	19.21	12.49	1.53
Dolphin	534,156	1,020,469	1.91	1.08	1.72	1.59	22.56	19.31	1.16
Hand	1,330,951	2,307,758	1.76	1.28	1.48	1.15	17.98	10.94	1.64

Table 4.1: Comparison of a fuzzy kD-tree that is build for the complete animation vs. ordinary kD-trees that are build per keyframe. The number of traversal steps, triangle intersection, and FPS are averaged over all keyframes and amortized for a single ray, as frustum kD-tree traversal is used. For not too complex scenes – like the Hand, Ben, and Dolphin model, the traversal steps and intersections only double at most for the fuzzy tree. For larger, and more complex scenes, e.g. Chicken and Cow, the statistics looks worse. But nevertheless, the important FPS performance does not follow linearly the traversal, and intersection statistics and at most an acceptable performance lost of a factor of 2.6 is observed for these test scenes.

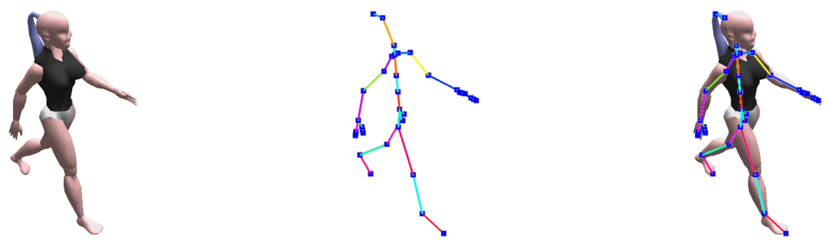

Figure 4.9: An example of a skinned model and its underlying skeleton. The colored lines show the skeletons bones and the blue squares are the joints that connect the bones to its skeletal structure.

tween two keyframes of the same animation sequence, and second to blend seamless two different animation sequences like a run and jump sequence of an avatar. Nevertheless, the restrictions that the keyframes have to have the same number of triangles and identical connectivity is still necessary. The details for the necessary motion decomposition as well as results will be discussed in the following sections.

4.5.1 Skinned Animations

After the verbal description of skinned animations in Section 4.2, Equation 4.5 formalizes the procedure. Consider a single vertex v whose position can be influenced by n bones. The skinned position v' can then be computed as a weighted sum from all weights w_i, affine transformation matrices $\mathrm{A_i}$ and the vertex v.

$$v' = \sum_{i=1}^{n} w_i \mathrm{A_i} v \tag{4.5}$$

In the general case, the weights w_i are specified by the application and normalized such that their sum is equal to one. For animated sequences it was shown in Section 4.4.1 how to compute the transformation matrices $\mathrm{A_i}$. Here, dealing with skinned animations, this is not necessary anymore because the application that uses the ray tracing engine has to specify how the animation should look like, and returns $\mathrm{A_i}$ and w_i to the ray tracer. Since a skeleton is given that guides the animation, the pure bone motion – which is in fact an affine transformation – can directly be used. No further computations are necessary e.g. to determine an optimal reference mesh. The reference mesh is simply an initial mesh that is defined by a modeler. All this features simplify the preprocessing to a great extend.

4.5.2 Mesh Segmentation

In Section 4.4.2 it was shown how a mesh can be decomposed into areas of coherent motion for animated sequences using a clustering approach. One observation at this point was, that if the models motion behaves like it has an underlying skeleton, the clustering process will identify these regions that map directly to the skeletons bones (see the Hand and Ben model in Figure 4.4 for an example). That means, a bone structure directly yields a usable clustering by itself. If each triangle can be only influenced by a single bone, there is directly a one to one mapping. However, this is not always the case. For triangles that are affected by multiple bones, it is not directly clear to which bone they belong best. To solve that problem, the fuzzy bounds can be computed for each bone (see Section 4.5.3) and a triangle is assigned to the one where it has the smallest residual motion.

4.5.3 Conservative Fuzzy-Bounds Estimation

Given a set of keyframes and Equation 4.5 it is clear that the maximum fuzzy-bound extends are not directly specified by the keyframes as it was the case with the animation sequences. The maximal extend can be in-between two keyframes. To make the motion decomposition work for skinned animations it is necessary to conservatively estimate those fuzzy-bounds. But too large and conservative fuzzy-bounds will decrease the ray tracing performance. If the fuzzy-bounds are too small rendering artifacts will occur. A conservative method is to sample the complete pose space that could be formed with the bones. Just rotate all bones around their joints, and all combinations of bone rotations, and compute the fuzzy-bounds for the triangles per bone. This will result in conservative but large fuzzy-bounds. To shrink the fuzzy-bounds it is also possible to exploit natural limits to a bones motion e.g. most people are not able to rotate their wrist around 360^o. If that kind of information can be obtained from an application, much "tighter" fuzzy-bounds can be computed.

4.5.4 Results

To evaluate the motion decomposition approach for skinned animations two datasets are used (see Figure 4.10 for a description) for the measurements. The core ray tracing engine is the same as described in Section 4.4.4. In order to add skinning support to the ray tracer, and to obtain all necessary information for the motion decomposition, the CAL3D open source library is used [Ope06]. All renderings are measured with a 1024^2 pixel screen size, simple diffuse shading, and an Opteron CPU that is clocked at 2.4GHz.

Fuzzy-Bounds Estimation

Skinned animations allow for a certain freedom in the potential motion of an object. As described above, dependent on the chosen freedom the fuzzy-bounds will be rather larger or small. In Table 4.2 the correlation between the motions freedom and performance is shown. When many arbitrary bone rotations are used for sampling the fuzzy-bounds it is clear that the fuzzy-bounds, and thus their overlap and average number of triangles per leaf, are large resulting in a decreased ray tracing performance. Applying some constraints, e.g. only natural/realistic bone movements are allowed, less samples are required and it can be observed that the fuzzy-bounds decrease. When more and more restrictions are applied this trend continues resulting in the best performance where the motion of just one animation sequence, consisting of several keyframes.

The optimal choice for the used sampling method is of course depended on the application. If the motion decomposition should be used in a 3D modeling package it is very likely that arbitrary bone rotations should be used for sampling. For computer games, it seems to be best to sample all possible animation sequences such that blend effects between them are possible. Nevertheless, at least for the tested animations, all sampling strategies result in interactive frame rates.

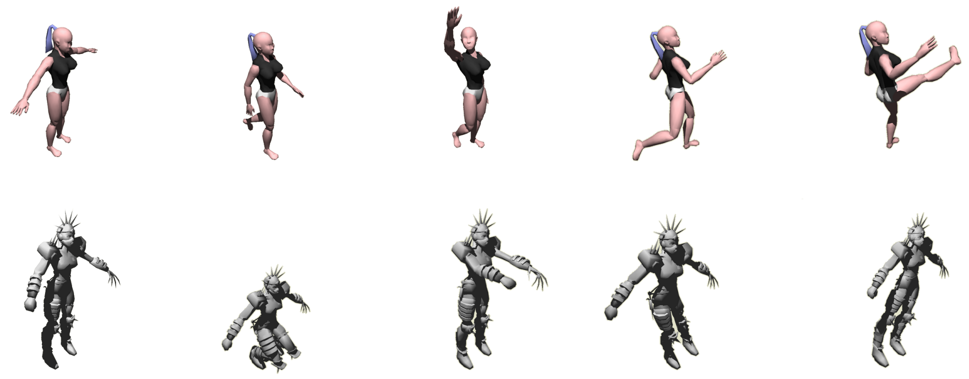

Figure 4.10: The two example models used for performance evaluation: Top, the Cally dataset. Cally consist of roughly 3k triangles and four skinned animation cycles (going, running, waving and kicking). Bottom, an avatar from UT2003. The UT2003 avatar consists of 2k triangles and has three skinned animation cycles (knee bend, crook of the arms, and swinging).

Pose Space Sampling	#Samples	Fuzzy Area	#Tris per Leaf	FPS
Arbitrary Bone Rotations	1000	747	4.1	7.7
Applying Joint Limits	245	580	3.4	10.2
Several Animation Sequences	124	552	3.2	11.6
Just One Animation Sequence	62	515	3.1	12.0

Table 4.2: Effects on restricting the sampled pose space: As the pose space sampling is restricted, the area of the fuzzy-bounds as well as the average number of triangles per leaf node decrease. Please note that even sampling arbitrary bone rotations results in acceptable frame rates. These would also allow to use the motion decomposition approach in the design phase of animation sequences.

Mesh Size Influence

Certainly, the size of the mesh affects the preprocessing as well as the rendering time. In Table 4.3 some important results are summarized. Probably the most important number is the preprocessing time for the sampling. As can be seen the sampling performance is almost linear in the number of triangles. But even for larger meshes with more than 200k of triangles the sampling time is below a minute. Compared to the preprocessing time of the animation sequences in Section 4.4.4 it can be noticed that the preprocessing time is greatly reduced while allowing a certain freedom in the objects motion at the same time. Also the time for constructing the fuzzy kD-trees stays in a

Preprocessing	Cally					UT 2003				
Time in Secs.	3k	10k	30k	90k	271k	2k	7k	22k	67k	200k
Sampling	0.63	1.80	5.21	16.40	50.40	0.59	1.23	3.08	8.63	29.7
kD-trees Build	0.65	2.28	8.77	34.50	150.00	0.39	1.67	6.41	23.8	88.9
FPS	9.50	7.10	5.70	3.60	2.00	15.30	12.30	7.40	4.00	1.90

Table 4.3: The influence of the mesh size to preprocessing and rendering performance: As the size of the meshes is further increased – by triangle subdivision – the sampling and kD-tree build stays in a reasonable time period. On the other hand the rendering performance suffers more as expected from a ray tracing system because the skinning operation is linear in the number of triangles. That explains the significant drop in rendering performance as the number of triangles increases.

reasonable time period as the number of triangles increases. Unfortunately, the overall rendering performance drops down significantly with an increased triangle count – more than expected. A reason is that the per frame skinning operations are linear in the number of triangles and thus the well-known logarithmic scaling with the scene size for ray tracing is lost. A way to avoid that penalty would be to use some additional *motion LOD* methods e.g. to simplify the mesh and skeleton as proposed in [AOW06]. Nevertheless, for realistic models, high interactive frame rates can be achieve.

4.6 Conclusions and Future Work

In this chapter a novel approach was presented that allows to ray trace efficiently keyframed animation sequences and skinned animations using hierarchical index structures. The main idea is to decompose the motion of an object into two independent parts: an affine transformation and some residual motion Δ. Since the animations are in general too complex for just a single transformation and Δ per keyframe, the meshes have to be decomposed into coherently moving regions. In order to make the motion decomposition work, several assumptions have to be made. The triangles connectivity of the animation has be constant across the animation, and the animations have to have some local coherent motion that can be exploited. If the animation has no local coherent motion the can be exploited the motion decomposition will still work, but the ray tracing performance will be very likely poor.

Although the number of traversal steps and triangle intersections are increased compared to an optimized index structure, for a particular pose in an animation, the rendering performance is always in an interactive range even when only a single CPU core is used. Typically the performance drops just by a factor of two. Furthermore, the required main memory requirements are moderate. For each cluster, or bone, one fuzzy index structure is required plus an transformation matrix for each keyframe when animation sequences are used for rendering. Currently, the largest memory overhead are the triangle meshes of the keyframes. If memory is here an issue, the animations could be compressed [AM00, KG04], and the required triangles decoded before an intersection test can take place. Additionally, the required preprocessing for both skinned meshes and animation sequences is sufficient fast and not cumbersome for practical applications. However, there is still room for improvements i.e. rather then sampling the pose space of skinned animations it might be possible to analytically compute the fuzzy-bounds. One of the real beauties of this algorithm is that a ray tracing implementation that already supports affine transformations, i.e. using Wald's [WBS03] two-level approach, do not have to change their core ray traversal and intersection code. These implementations can with just a few modifications render additional classes of dynamic scenes without the need to change complex and critical parts of code.

In the last two years a true flood of approaches for dynamic ray tracing has been published. To name just a few there are: new algorithms for fast kD-tree [PGSS06, WH06] and BVH [Wal07, LYTM06] construction, fast BVH updates [YCM07], selective BVH updates [YCM07], asynchronous BVH updates [IWP07], and fast grid construction methods [WIK$^+$06, ISP07]. This list is far from complete. Right now it is not possible to compare all of those approaches since they all support different kinds of motion, have a different code bases, use internally different algorithms, e.g. for ray traversal or intersection tests, and data structures [WMG$^+$]. One possibility for future work could be to concentrate on particular domains where specific knowledge is available to determine a (close to) optimal index structure and guide its (re)builds, updates and traversal algorithms that depend on the characteristic of the application. Finally, it can be said that the presented motion decomposition approach is for typical scenes on par with other solutions, in terms of memory overhead and performance, that could be used for computer games or animated movies.

Chapter 5

Ray Tracing on the CELL Processor

In Chapter 3 it is discussed that hardware support for ray tracing is essential to achieve sufficient performance for 3D-based computer games. In this chapter the processor of the Playstation 3, the IBM Cell processor, will be analyzed in terms of ray tracing performance. Specifically, it is interesting to explore what kind of algorithmic changes and extensions are needed to achieve the best possible ray tracing performance, and what the inherent limitations are for this new processor architecture.

5.1 The CELL Broadband Architecture

Before the implementation of the ray tracing algorithm to the Cell processor is discussed, first the unique features of this architecture are described. In particular where it differs from traditional CPU architectures. One of the most important differences to conventional multi-core CPUs is that the Cell is not a homogeneous system with multiple copies of the same core [KDH$^+$05a]. Instead, it is a heterogeneous system, consisting of one 64 bit PowerPC core (PPE) and eight *synergistic processor elements* (SPEs), each of which contains a *synergistic processing unit* (SPU) (see Figure 5.1). The SPE differs from standard x86 CPU cores, in that it is much smaller, exclusively designed for streaming workloads, and do not support hardware multithreading. In its intended use, the Cell's SPEs do the *real* work in a data parallel or streaming (pipeline) manner, while the PPE core performs synchronization tasks and executes non-parallelizeable code.

Instead of a memory cache, each SPE has 256KB of *local store* memory. Direct access to the PPEs main memory from an SPE is not possible. Instead the SPE can explicitly perform (asynchronous) DMA transfers to or from main memory. The local store has to accommodate both data and code. The SPEs have their own reduced RISC-like instruction set, where each instruction is a 32 bit word with a fixed execution latency of two to seven cycles (except double precision floating-point instructions that are not used and hence not considered here).

Compared to standard x86 CPUs, each SPU has a large unified register file of 128×16 byte SIMD registers. Most of the SPE instructions are SIMD instructions, in particular for performing multimedia and general floating-point processing. These instructions are more flexible than e.g. Intel's SIMD instruction sets [Int02b] and include, for example, three-operand instructions with a throughput of one SIMD multiply-add per cycle, allowing for a theoretical peak performance of 25.6 (single-precision) GFlops on a 3.2GHz SPE. Each SPE has two pipelines, where each is specialized for a certain type of instructions (load/store vs. arithmetic). This allows for dispatching two (independent) instructions in parallel, achieving a theoretical throughput of 6.4 billion SIMD instructions per second per SPE. Besides the special instruction set, the SPE has no branch prediction as known from standard CPU cores. Instead a branch hint instruction is supported which helps the SPE to predict if a certain branch has to be taken or not. The branch hint takes a *branch target address* – that should point to the most likely executed branch – and prefetches up to 32 instructions starting from the target address. In doing so, a correctly predicted branch does not incur any penalty.

Both, PPE and SPEs are in-order processors. As the 256KB local store of the SPE has a fixed access latency of 7 cycles, in-order instruction execution is a suitable simplification: The compiler can predict the (local-store) memory access and therefore schedule the instructions for maximum performance. However, in-order execution on the PPE with its standard 32KB L1 and 512KB L2 cache harms memory intensive applications, making the core less powerful compared to standard x86 CPUs with out-of-order execution. In order to reduce the impact of cache misses, the PPE uses 2-way symmetric multi-threading [KDH+05a] which is comparable to Intel's Hyperhreading [Int02c].

To keep the SPEs supplied with data – and to allow efficient communication between the SPEs – the Cell uses a high performance *element interconnection bus* (EIB). The EIB is capable of transferring 96 bytes per cycle between the different elements – PPE, SPEs, I/O interface, and system memory. The DMA engine can support up to 16 concurrent requests per SPE, and the DMA bandwidth between the local store and the EIB is 8 bytes per cycle in each direction. The aggregate bandwidth to system memory is 25.6 GB/s, and the bandwidth between SPEs can be more than 300GB/s on a 3.2GHz Cell. SPE-to-SPE transfers are kept within the EIB, avoiding any main memory transaction.

The minimalistic design of the SPEs (no cache, no branch prediction, in-order execution, no hardware threading support) allows for very high clock rates. For experiments a dual Cell-Blade system (with 512 MB of XDR main

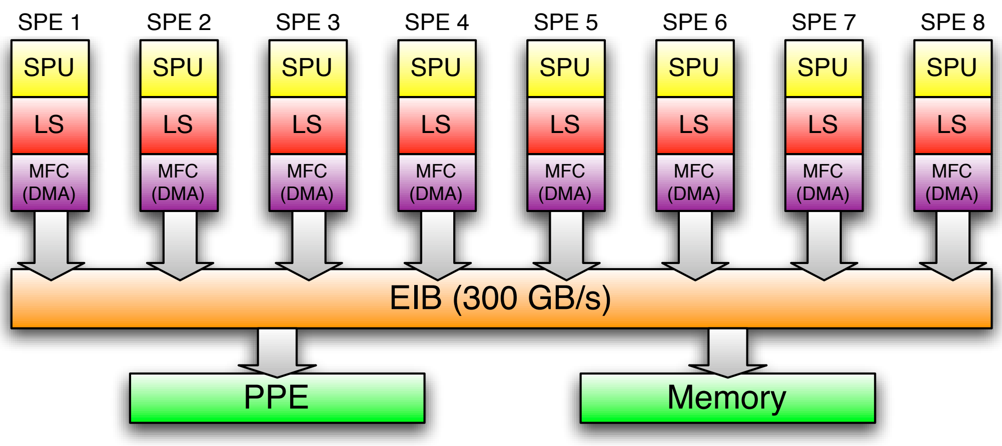

Figure 5.1: Each Cell consists of a 64 bit PowerPC core (PPE) and eight *synergistic processor elements* (SPEs). Each SPE has 256KB local store, a memory flow controller (MFC) and a *synergistic processing unit* (SPU) with a SIMD processing unit and 128 registers of 16 bytes each. An *element interconnection bus* (EIB) with an internal bandwidth of more than 300 GB/s (per 3.2GHz Cell processor), is responsible of transferring data between the SPEs. The maximum bandwidth from the SPEs to main memory is 25 GB/s.

memory) is used in which the Cells are clocked at 3.2GHz (which is the same clockrate as the PS3 offers).

5.2 Ray Tracing on the CELL Broadband CPU

As just shown, the Cell is quite different from today's x86 based mainstream processors. In order to enable efficient ray tracing on the Cell, the following differences have to be taken care of (Section 5.9 discusses also the drawbacks that are inherent to the current Cell architecture):

In-order Execution: An SPE executes the instructions in-order, which means that pipeline stalls, caused by code dependencies or mispredicted branches, are more expensive than on a CPU with out-of-order execution. To avoid this, the compiler is responsible for a suitable instruction scheduling and to untangle code dependency chains. Most of the time the compiler resolves the dependencies automatically, but sometimes the algorithms have to be (manually) adapted to help the compiler finding independent instruction sequences. These instruction sequences can then be interleaved to prevent stalls efficiently.

SIMD Instruction Set: As the SPE's instruction set is designed for SIMD processing, most of the instructions operate on multiple data elements at once (two to sixteen elements depending on element size). As an instruction has a throughput of one per cycle and a latency between two to seven cycles, one has to ensure enough independent data to work on. Otherwise, dependency chains, and therefore pipeline stalls, are unavoidable. Unfortunately, the instruction set is suboptimal for scalar code, so even simple operations such as increasing an unaligned counter in memory require a costly read-modify-write sequence.

Memory Access: Each SPE has an explicit three-level memory hierarchy: a 128×16 bytes register file, a 256KB local store, and main memory. As the local store does not work as hardware-managed memory cache, all main memory accesses must be done explicitly by DMA transfers. Even though the memory bandwidth of 25.6 GB/s is rather high, each memory access has a high latency of several hundred SPE clock cycles. In order to hide the latency, the DMA engine supports asynchronous transfers, whose states can be queried on demand. Even though this setting is ideal for streaming operations in which huge blocks of data are being processed sequentially, it is challenging for a data-intensive application with irregular memory accesses such as a ray tracer.

Parallel Execution: Each Cell has 8 SPEs. There are different ways of mapping an algorithm onto such a parallel architecture, and the exact way this is done will have a significant impact on performance.

In the next section these mappings and its possible implications will be discussed in detail.

5.2.1 Ray Tracing Algorithm Mapping

The first design decision to consider is how to map the ray tracing algorithm to the multiple SPEs. In a heterogeneous approach, each SPE runs a different kernel, and the results are sent from one SPE to another. In a homogeneous approach, each SPE runs a full ray tracer, but on different pixels. Traditional multi-core architectures favor the latter, but traditional streaming architectures are usually intended to be used in the heterogeneous way (also see [KDK+01, PBMH02, CDR02]).

As seen in Chapter 2, ray tracing can be broken into the tasks: ray generation, traversing rays through a spatial index structure, intersecting the rays with geometric primitives, and shading the corresponding intersection points, including the generation of secondary rays and their recursive evaluation. One way of mapping ray tracing to the Cell is to have each SPE perform only one of these tasks, and to send its results to the SPE that performs the next task in the ray tracing pipeline. For example, one SPE could generate primary rays which are then sent to one or more SPEs doing the traversal, which in turn send ray-triangle intersection tasks to other SPEs. In fact, the Cell's architecture is able to support such a streaming workload: the high inter-SPE bandwidth (of up to 300GB/s) lets the SPEs communicate with each other; and the asynchronous DMA transfers allow for transferring one SPE's output to another while both operate on the next block of data in the stream. In principle, mapping a ray tracer to such an environment is possible, and has been demonstrated for Smart Memories [CDR02] and GPUs [PBMH02]. The streaming approach works best if the individual tasks can be cascaded and if there is a steady flow of data from one task (i.e. SPE) to the next (as in video/speech processing, or scientific computations) [WSO$^+$06]. A ray tracer, unfortunately, has a much more complex execution flow: the traversal unit does not pass results unidirectionally to the intersection unit, but also has to wait for its results; the shader not only shades intersection points, but can also trigger additional rays to be shot; etc. Such dependency chains require one task to pause and wait for the results of another, creating stalls. In addition, this approach makes it hard to balance the load of different SPEs, as the relative cost of traversal, intersection, shading, etc., varies from pixel to pixel. As soon as one SPE in the chain becomes a bottleneck, it starves the other ones. To a certain degree this starvation can be avoided by buffering the SPEs in- and outputs in main memory, and then frequently switch the kernels each SPE executes depending on what tasks need to be done most. This implies a non-trivial system design (synchronization, load balancing, etc), and also poses significant strain on the memory system (whose bandwidth is more than an order of magnitude lower than the inter-SPE bandwidth). Even though 25 GB/s seem plentiful, the target is to achieve several dozen frames per second, each frame requiring at least one million rays, and reading/writing each ray several times when passing it from task to task may easily create a bottleneck.

The above considerations have led us to follow an approach typically used on conventional shared-memory multiprocessor architectures: each SPE independently runs a full ray tracer, and parallelization is achieved by SPEs working on different pixels (see Section 5.7). Having each SPE work independently ensures less communication between SPEs, and avoids exchanging intermediate results with either other SPEs or main memory. In addition, it avoids dependency chains between different SPEs in- and outputs, and facilitates high SPE utilization. On the downside, having each SPE run a full ray tracer forces us to operate the SPEs in a way they are not designed to be used: in particular, each SPE may access any data in the scene database, in a random-access manner. As the local store is too small to store the entire scene, this requires appropriate ways of accessing and caching the scene data, as well as means to handle the resulting SPE-memory dependencies.

It also limits the possible code size for each pipeline stage size drastically. All algorithms from ray generation to shading have to share the same small local store.

5.2.2 Spatial Index Structure and Traversal Method

Having decided on the programming model, the next decision is which spatial index structure to use. As discussed in Section 2.4.2 efficient ray tracing requires the use of spatial index structures, such as bounding volume hierarchies (BVHs) [RW80], Grids [AW87], or kD-trees [Jan86]. In particular, tracing coherent packets of rays [WSBW01] – possibly accelerated by looking at the packet's bounding frustum [RSH05] – has been shown to be an important factor in reaching high performance [Wal04, Ben06]. As explained earlier, tracing packets of rays allows for amortizing memory accesses over multiple rays, allows for efficiently using SIMD extensions, and increases the compute-to-memory access ratio. Though already important for a conventional CPU, these advantages are even more important for a Cell, which depends on dense SIMD-code, and for which memory accesses are even more costly than for a standard CPU. Measurements have shown that a single DMA transfer requires about 1000 clock cycles until it is finished.

Naturally, since computer games are the main focus of this thesis, the acceleration structure of choice should also support dynamic scenes. Among the three discussed traversal acceleration structures in Section 2.4.2 the Grid is the most general in the kind of animations it supports. Its more regular structure would nicely fit a streaming architecture. For example, a straightforward extension of the technique proposed in [MFT05] would allow for prefetching the grid cells before traversing them. However, the BVH and kD-trees currently seem to be faster than the Grid, more suitable for complex scenes – in particular for scenes with different geometric density –, and somewhat more robust for secondary rays [WBS07]. Since the BVH can support the same kind of animations, as a kD-tree even without the preprocessing discussed in Chapter 4, the BVH is chosen (with 8×8 rays per packet), but most ideas generalize to Grids and kD-trees as well. Compared to a kD-tree, a BVH offers offers even more advantages that are particularly interesting for a Cell like architecture: a BVH will have fewer memory accesses and a higher compute-to-memory access ratio, because it has fewer nodes than a kD-tree, and more arithmetic to be done per node.

5.3 CELL-Specific Traversal and Intersection

As mentioned in Section 5.2.2, this system closely follows the traversal proposed in [WBS07], and the traversal algorithm and triangle intersection are exactly the same. Nevertheless, the Cell is not like the CPUs that the original traversal and intersection framework was designed for. Therefore, special optimizations have to be done to efficiently map these routines to an SPE.

5.3.1 BVH Traversal

Branch mispredictions on the Cell are costly. Unfortunately, the BVH traversal proposed in [WBS07] (see also Section 2.4.3) has two conditionals in its inner loop, both of which have a 40-45% chance of being taken: to reduce ray-box tests, one does not test each ray against every node, but first performs two tests that can often decide the traversal case without having to look at all the individual rays. First, one tests the first ray that hit the parent node, and immediately descends if it hits; if not, the node is tested against the packet's bounding frustum, leading to an immediate exit if the frustum misses the box. These two tests cover around 80-90% of the traversal cases, making an efficient implementation mandatory. As a serial execution of the two tests introduces dependency chains and therefore pipeline stalls, two tests are performed in parallel, while postponing the branches as far as possible. Moreover, the branches are arranged in such a way, that mispredicted branches occur only for the third traversal case. The parallel computation completely avoids dependency stalls and increases the double instruction dispatch rate to 35%, yielding total costs of 51 cycles (without a misprediction stall), which is an 10-20% performance improvement, compared to a serial test execution. The average cycles-per-instruction (CPI) ratio for the code is 0.65, where 0.5 is the optimum (please remember that each SPU has two pipelines and can thus dispatch, under some circumstances, two independent instructions as described in Section 5.1).

5.3.2 Triangle Test

As packet-triangle test, the algorithm proposed in [Wal04, Ben06] is used: this test is particular suited for SIMD processing, and in addition, stores all data required for the triangle test in a single memory location (see Section 2.4.4). As proposed in [WBS07], a SIMD frustum culling step is performed to detect if the frustum completely misses a triangle.

As with the BVH traversal, the triangle test was originally designed for a x86 CPU, and the Cells in-order execution model requires some changes to remain efficient. The triangle test consists of four individual tests: one first tests the distance to the triangle's embedding plane, and then computes and tests the three barycentric coordinates of the point where the ray pierces the plane. On a x86 CPU, the best performance is achieved if each of these tests is immediately followed by a branch that skips the remaining computations if it failed. On the Cell, these branches cause dependency chains and frequent branch misprediction stalls, which cause the same code to run quite inefficient. To avoid these, all branches are removed, and always sixteen SIMD tests (i.e. for 64 rays) are performed in parallel. Results are updated via branch-free conditional moves. With these modifications, a packet of 64 rays can be intersected in only 520 cycles, or 8.125 cycles per ray on average. In particular the Cell's large number of registers is quite helpful: while on an x86 CPU even a single test can lead to register spilling, the SPE's 128 registers allow for unrolling the intersection test eight times, which yields a double instruction dispatch of 37.5% and a CPI ratio of 0.71. Moreover, all required triangle data can be held in registers, without having to reload – and reshuffle – the triangle data for every new batch of rays in the packet.

5.4 Explicit Data Caching

In Section 5.1 it is stated that the Cells SPE's do not have a cache memory but local store. What does that mean? Since the advent of the Von-Neumann architecture, CPU's clockrate increases at a higher pace as access times to main memory decrease. This so called Von-Neumann bottleneck [Hil89] is steadily increasing as more and more clock cycles are wasted for slow data request from main memory and the CPU has nothing to do but wait. For this reason, traditionally CPUs have a small, yet expensive, low latency memory (hierarchy) on-die. In this on-die memory data from main memory can be transfered, and computations using the data that is already "on-the-chip" can be performed with highly reduced data access times. Unfortunately, the size of the the small on-die memories is way too small for all code and program data that are needed for most applications. A cache is now a combination of an on-die memory and logic that automatically fetches requested data into the cache, if not already residing there, and decides which data has to be swapped out if the cache is full. The SPE's local store is missing the cache logic and data have to be swapped in and out manually.

5.4.1 On Caching

The cache has the primary task to cache the data that are most likely reused soon. In general a cache consists of two parts: a *tag* list and associated *data* blocks. The tag is a *base address* where in the main memory the data can be found and is thus a unique identifier. Data blocks – also called cache lines – have typically a size of 2^n bytes, e.g. 64. The size of the cache lines determines the granularity of memory requests. If a program fetches some data, the request-address is translated to a tag, and looked up in the tag list. The translation is necessary to also find cached addresses in the tag list whose address lies within a cache line. When the tag can be found, the data is already in the cache and can be accessed in the associated data block of the tag. Otherwise the data, more precisely a complete data block starting

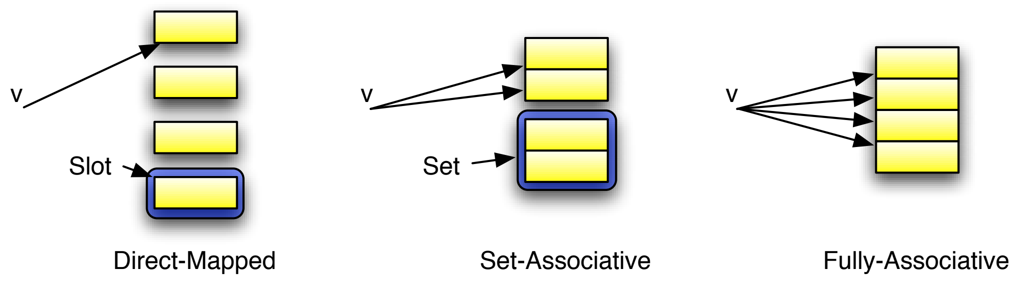

Figure 5.2: The three standard cache organizations: In a direct-mapped cache, the date of a particular address can only be stored in one slot. There exists then a direct 1:1 mapping from main memory addresses to the slots in the cache. In a set-associative cache, slots are grouped to sets. Addresses are then mapped to sets. The date of an address can then be placed in each slot of a set. Fully-associative caches allow to store the date of an address to every slot.

from the tag address, is requested from main memory, and the tag is placed in the tag list.

There are three standard ways a cache can be organized [Bre03] (see Figure 5.2). First there is the fully-associative cache. In this cache, the data of an address can be placed in every slot in the cache. To determine if some data is in the cache it is necessary to check all tags whether the tag is found. If the cache is reasonable large, e.g. it has 256 entries, this linear search is very expensive. Contrary to a fully-associative cache, in a direct-mapped cache there is for every address only **one** slot where its data can be placed. This makes the cache look up very cheap since the tag has only to be searched in one slot. Nevertheless, multiple addresses map to each particular slot and data will be often replaced by other data that map to the same slot in the cache. A compromise between these two cache organizations is the set-associative cache. Here, sets of slots are grouped together e.g. four. In that case the cache could be refered as a 4-way set-associative cache. The advantage is that addresses are now mapped to sets, and that the data can be placed in any slot – the so-called ways – within a set. A fully-associative cache is in that sense a set-associative cache with only one set containing all slots. If a set-associative cache is implemented in software it is advantageous to choose 4-ways. The four tags of a set can then be placed in a SIMD-register and a requested tag is simply looked up by a single SIMD-compare.

To compute now exactly the tag, set index, and byte offset in a cache line for an address, only some bit operations are necessary if the size of a cache line and the number of sets is a power of two. Lets consider a 4-way

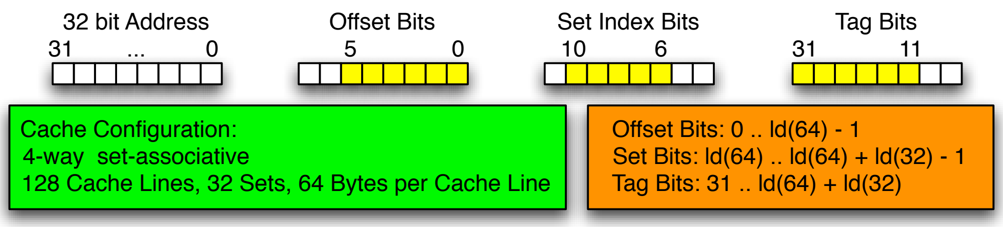
32 bit Address
31 ... 0
Offset Bits
5 0
Set Index Bits
10 6
Tag Bits
31 11
Cache Configuration:
4-way set-associative
128 Cache Lines, 32 Sets, 64 Bytes per Cache Line
Offset Bits: 0 .. ld(64) - 1
Set Bits: ld(64) .. ld(64) + ld(32) - 1
Tag Bits: 31 .. ld(64) + ld(32)

Figure 5.3: An example for the computations of the offset, set index, and tag of a 32 bit address.

associative cache with 128 cache lines ($s = 32$ sets) with a capacity of $b = 64$ bytes per data block. The offset of an address in a cache line is determined by its least significant bits from 0 to $\log_2(b)$. To find the set index, the bits from $\log_2(b)$ to $\log_2(b) + \log_2(s) - 1$ can be used. Finally the tag can be extracted from the most significant bits from 31 to $\log_2(b) + \log_2(s)$ – on a 32 bit machine. Figure 5.3 clarifies the address mapping with an example.

But what happens now when data have to be overwritten because there are already data in a slot or the cache is completely full? In the case of the direct-mapped cache new data always overwrite old data. There is no other way since an address only maps to one particular slot. Set- and fully-associative caches need a replacement strategy that decides what data should be overwritten. The goal of the replacement strategy is to leave the data in the cache that are very likely to be reused in future and overwrite that data that are not needed anymore. That is in general a very difficult problem since it cannot be known in advance what data will be needed in future. For example a simple rule could be used to evict the data in a set's slot that is not used for the longest time.

Typically on todays commodity CPUs caching is performed in hardware. Nevertheless, one of the SPE's design goals was to use as little as possible die-area and thus hardware caching capabilities were considered as too expensive. However, each SPE requires access to all scene data, which does not fit into local store. As main memory accesses can only be performed by DMA transfers, caching is emulated by creating small self-maintained software memory caches within the SPE's local store. The lack of a hardware supported cache means that all cache logic has to be performed, using serial code, in software – which is costly. In addition, cache misses result in high-latency DMA transfers, and are also quite costly. Finally, even cache hits require a short instruction sequence to obtain the data, increasing access latency of cached data. Fortunately, caching of scene data in a ray tracer has shown to yield high cache hit rates [SWW$^+$04, SWS02, WSS05] and cache accesses can be additionally amortized over an entire ray packet.

5.4.2 Cached Data Types

Instead of a unified memory cache, an approach recently used for designing ray tracing hardware [SWS02, WSS05] is followed that uses specialized caches for each kind of scene data. Having specialized caches allows for fine tuning each individual cache's organization and granularity. Due to the DMA transfer granularity, all cache granularities must be powers of two. For the kD-tree based hardware architectures, three types of caches are required: node caches, triangle caches, and item list caches. A BVH references each triangle exactly once, and item lists can be completely abandoned.

Node Cache: Each BVH node stores minimum and maximum box extent, a pointer to the first child (for inner nodes) or first triangle (for leaves), and some additional bits. These can be stored within a 32 byte data structure, which is also a power of two. In addition, a BVH – in contrast to a kD-tree – always has to test both children of a node (see Section 2.4.3). Due to that, individual BVH nodes are not cached, but instead use a 64 byte granularity and cache pairs of BVH node siblings. Compared to caching individual BVH nodes, this yields a roughly 10% higher cache hit rate.

Triangle Cache: As mentioned above, the triangle test proposed in [Wal04] is used, which uses a precomputed record of 12 words (48 bytes) for the triangle test. This data is fully sufficient for the intersection test, so no additional caches for triangle connectivity or vertex positions are required, which greatly simplifies the cache design. As 48 is not a power of two, a triangle cache's cache line size of 64 bytes is chosen. The additional 16 bytes are then used to store indices to the three vertices respectively normals and an index to the global shader list. These are not required for the intersection test, but are required when shading the intersection points and hence, additional costly DMA transfers can be avoided.

5.4.3 Cache Organization and Efficiency of Caching

Ray traversal offers a high degree of spatial coherence, and even a simple direct mapped cache offers high cache hit rates (see Table 5.1). A four-way set-associative cache (with a least-recently-used replacement policy) provides an additional 1-5% higher cache hit rate but requires significantly more complex logic, which increases cache access latency. As a cache access has to be performed in every traversal step, better performance can be achieved with a direct mapped cache, even though it has a somewhat lower hit rate. Fully-associative caches were not considered since cache look ups are way too expensive. Though a BVH instead of a kD-tree is used, the cache statistics in Table 5.1 show nearly identical results to those reported in [SWS02, WSS05], showing similarly good cache hit rates. In order to reduce instruction de-

	Cached	128KB		256KB		512KB		1024KB	
Scene	Data	4-way	DM	4-way	DM	4-way	DM	4-way	DM
ERW6	BC	99.8	99.4	99.9	99.7	99.9	99.9	99.9	99.9
Conference	BC	95.7	91.2	98.0	94.4	98.5	96.9	98.7	97.8
VW Beetle	BC	87.6	84.4	91.6	88.3	93.5	91.6	94.1	93.2
ERW6	TC	98.1	96.8	98.4	97.4	98.7	98.4	98.7	97.7
Conference	TC	71.0	61.5	80.0	74.8	86.7	82.7	98.2	86.5
VW Beetle	TC	45.1	39.8	50.9	45.8	55.1	50.4	57.2	52.9

Table 5.1: Cache hit rates for 4-way associative (4-way) vs. direct mapped (DM) caches for BVH nodes cache (BC) and triangle cache (TC). Measured with casting primary rays at a resolution of 1024^2 pixels, 8×8 rays per packet, and a default setting of 256 BC entries (17KB), and 256 TC entries (17KB).

pendency chains, speculative execution is applied: In parallel to the cache hit test, the data is speculatively loaded from the cache and reformatted for further processing. The potential branch to the miss handler is slightly postponed, which allows for hiding cache access latency by interleaving the instruction sequence with surrounding code. As the cache hit rate is typically very high, the increased number of instructions executed in case of a cache miss does not have a significant impact. Due to the high cache hit rates, branch hints are used to optimize all cache access branches for hits. Thus, the hit logic is cheap, and a costly branch miss occurs only in the case of a cache miss.

5.4.4 Cache Sizes

In addition to the caches, the SPE's local store must also accommodate program code, ray packet data (rays and intersection points), and some auxiliary buffers. Since local store is scarce, the cache sizes must be chosen carefully. Table 5.1 shows that for three test scenes a BVH node cache of 256 entries is a good compromise between cache hit rate ($> 88\%$) and memory consumption (17KB); doubling the cache size increases hit rates by a mere 3%, but doubles memory consumption. For the triangle cache, the situation is more complicated. Since triangle intersections are performed where the rays are least coherent (at the leaves), the triangle cache has a much lower hit rate, hence a large cache is beneficial. Nevertheless, even where the hit rates are very low–down to 40-55% for the Beetle scene – even a much larger cache cannot significantly improve the hit rate: for finely tessellated geometry, triangles are smaller than the spatial extent spawned by the 8×8 rays in a packet, and will therefore often be intersected by a single packet only. It can be argued that a ray bundle itself acts as a kind of zero-level cache with a very high cache hit rate even for fine tessellated scenes across the rays in a bundle. Even though the triangle cache's hit rates of only around 50% look utterly devastating, triangle accesses are rare compared to BVH traversal steps, so the total impact of these misses stays tolerable. Because of the costly DMA transfers, a cache hit rate of 50% still ensures a higher performance than using no cache at all.

5.4.5 Traversal Performance including Caching

With cached access to scene data, the per-SPE performance of the traversal and intersection code is now evaluated on three example scenes with different geometric complexity (see Section 5.8 for a detailed description). Table 5.2 gives performance data per SPE, casting only primary rays (no shading operations are applied). DMA transfers invoked by cache misses are performed as blocking operations, making cache misses quite costly. For the rather simple ERW6 scene – which also features very high hit rates – a single SPE (clocked at 3.2GHz) achieves 30 million rays per second(MRays/sec), the more complex conference and VW beetle scenes still achieve 6.7 MRays/sec and 5.3 MRays/sec, respectively.

5.5 Software Multithreading

Though a set of small self-maintained caches within the local store allows for efficiently caching a large fraction of the scene data, having only comparatively tiny caches of 256 entries (for more than half a million triangles), cache hit rates in particular for the triangle cache drop quickly with increasing geometric complexity. Being a streaming processors, the Cell is optimized for high-bandwidth transfers of large data streams, not for low-latency random memory accesses. All memory accesses are performed via DMA requests, which have a latency of several hundred SPE cycles. The discrepancy between bandwidth and memory latency is not a phenomenon unique to the Cell processor, but exists similarly for every one of today's CPU architectures. One of the most powerful concepts to counter this problem is software multithreading: the CPU works exclusively on one thread as long as possible, but as soon as this thread hits a cache miss, it is suspended, the data is fetched asynchronously, and another thread is being worked on in the meantime. If thread switching is fast enough, and if enough threads are available, multihreading can lead to a significant reduction of pipeline stalls and can therefore lead to higher resource utilization.

Though multithreading is most commonly associated with CPUs, it is also used in other contexts. For example, the RPU [WSS05] architecture makes

Scene	# TravSteps/ Bundle	# Isecs/ Bundle	Traversal (in %) Early Exits	Early Hits	MRays/sec
ERW	18.73	1.47	44	52	30.52
Conference	55.33	5.94	40	52	7.24
VW Beetle	43.9	7.21	34	48	7.08

Table 5.2: Performance per 3.2Ghz SPE, in frames per second for casting primary rays (no shading) at a resolution of 1024^2 pixels, with 8×8 rays per packet, 256 BVH cache entries (17KB) and 256 triangle cache entries (17KB). Triangle intersections means triangle intersections after SIMD frustum culling. Even though only a small amount of local store is reserved for caches ($<$ 35KB), and all DMA transfers are performed as blocking operations, a single SPE achieves a performance of 7 - 30.5 million rays per second.

heavy use of multithreading, and uses 32 simultaneous threads per RPU core to hide memory latencies. Similarly, the same concept has been used in Wald et al. [WSB01], albeit one level higher up in the memory hierarchy: instead of switching on a memory access, the system in [WSB01] switched to a different packet if a network access was required. Other systems use similar concepts (e.g. [PKGH97]). Though the Cells PowerPC-PPE does support multihreading, the SPEs do not. Still, similar to [WSB01] the concept can be emulated in software. Having no hardware support for the context switch, a complete SPE context switch which would include saving all registers and the complete local store to memory, would be prohibitively expensive. Therefore, a lightweight thread is defined as a single 8×8 ray packet, and multiple of them are traversed simultaneously. Thus, only a small data set has to be saved and restored. DMA transfers can be declared as non-blocking (i.e., asynchronous) and their state can be requested any time. Each time a cache miss occurs, an asynchronous DMA transfer is invoked and the traversal continues execution with the next packet; once the original packet is resumed, its data will usually be available. Listing 5.1 shows the complete procedure.

5.5.1 Implementation

In addition to the ray and intersection data, each packet also requires its own stack. Due to scarcity of local store, only a limited number of packets can be kept at the same time. In the current implementation, four packets are being used simultaneously. In order to suspend and to resume ray packets, all packet-specific data – the *ray packet context* – has to be saved and restored. In the current implementation, the packet context comprises a pointer to the corresponding ray packet, a stack pointer, a DMA transfer state, etc.

```
packetIndex = 0;
goto startBVHTraversal;

processNextPacket:
do {
   packetIndex = (packetIndex+1) \% NUM_SMT_PACKETS;
}
while(terminated[packetIndex] == false)

RestoreContext(packetIndex);

startBVHTraversal:
while(1) {
   if(stackIndex == 0) break;
   while(1) {
      index = stack[--stackIndex];
      if(InsidelocalStoreBVHCache(index) == false) {
            SaveContext(packetIndex);
            InitiateDMATransfer(index);
            goto processNextPacket;
      }
      box = GetBoxFromLocalStoreBVHCache(index);
      if(EarlyHitTest(box) == false) {
            if(RayBeamMissesBox(box) == true
               && AllRayPacketsMissBox(box) == true)
            goto startBVHTraversal;
      }
      if(IsLeaf(box) == true) {
            break;
      }
      else {
            stack[stackIndex++] = GetBackChildIndex(box);
            index = GetFrontChildIndex(box)    ;
      }
   }

   PerformRayTriangleIntersectionTests(box);
}
```

Listing 5.1: Pseudo-code for BVH traversal with software-multihreading. Once a cache miss occurs, the current context is saved, an asynchronous data transfer is invoked, and the traversal continues by restoring the next (not yet terminated) packet.

Pointers can be represented as 32 bit integers, and pointers for the four contexts can be stored within a single integer vector, which allows for quick insertion and extraction of data.

5.5.2 Performance Influence

As can be seen in Listing 5.1, software-multithreading is a non-trivial task, and the context saves and restores carry some significant cost as well. Still, this cost is lower than the several hundred cycles that would be incurred by waiting for the memory request to complete. Overall, software-multithreading (SHT) gives a noticeable speed up, as can be seen in Table 5.3, which compares the performance of a caching-only implementation (see Section 5.4.5) to the performance achieved when applying SHT to both BVH and triangle cache. As expected, SHT cannot give a noticeable benefit for small scenes in which only few cache misses occur anyway. For larger scenes, however, where cache misses become significant, SHT can achieve more than 30% improvement in performance.

5.6 Shading

Once being able to trace rays, the resulting intersection points have to be shaded. Ideally, the Cell would be used as a ray tracing processor only, with shading being done on a GPU. In a Playstation 3, for example, GPU and Cell have a high-bandwidth connection, and sending rays back and forth would probably be feasible. In that setup, the GPU could do what it's best at – shading – and the Cell would only trace rays. Nevertheless, a Playstation 3 can only use this high-bandwidth connection using its own operating system and libraries. Until now, it is not possible to use the available Linux environment with this feature.

In the following, a set of 8×8 intersection points is defined as an *i-set*. Each intersection point within an i-set comprises the triangle index, the hit point in world space, the shading normal (interpolated from the three vertex normals), the reference to a surface shader etc. For the SPE's SIMD architecture, shading is most efficient if multiple intersection points are shaded in parallel. Unfortunately, neighboring rays may have hit different geometry, requiring different data to be shaded. Since the smallest SIMD-size is four, these 8×8 intersection points are grouped into 16 intersection packets of four rays each, and work on each of these in a SIMD manner.

Compared to ray packet traversal, parallel shading has a much more complex control flow, and a significantly more complex data access pattern. In particular, while traversal always intersects all rays with the same node or triangle, shading each ray may require different shading data, which may have to be fetched from completely different memory locations (material data, ver-

	FPS		
Scene	w/o SMT	with SMT	Speedup (in %)
ERW6	30.52	30.54	0.01
Conference	7.24	8.27	14.2
VW Beetle	7.08	9.29	31.21

Table 5.3: Impact of software-multihreading (SMT) on per-SPE performance. Performance in frames per second (1024^2 pixels, 256 BVH node and triangle cache entries each, no shading) on a 3.2Ghz SPE. As scene size increases, it can be realized that SMT becomes more and more important.

tex positions, vertex normals, etc.). Though, in principle these accesses could be completely random, in practice there is at least some degree of coherence. For instance, neighboring intersection points typically have the same shader (even if they hit different triangles), and neighboring rays even frequently hit the same triangle. Although, of course, there is a possibility that the worst case scenario happens where all four rays have hit a surface with a difference shader, but this is very rare in practice.

Since the SIMD-width is four – not 64 – four rays are shaded always at the same time. For each intersection packet a flag whether the four intersection points refer to the same triangle are stored additionally. This allows for a more efficient implementation because cache accesses to the scene data can be amortized over the whole intersection packet. Smooth shading typically requires a surface normal that is interpolated by the three vertex normals, so an additional cache for vertex data is maintained while filling in the i-set. All vertex data – normal, position, and texture coordinates – is stored within a 64 bytes element, allowing to cache all vertex data in one aligned cache record. In addition to vertex data, a cache for material data (diffuse and specular color, etc.) is maintained.

The actual shading process is split into several steps. First, is is checked (by testing the triangle flag) whether rays in the packet have hit the same triangle, using the information to efficiently gather geometric data, in particular, the three vertex normals: the data is loaded once, and then (possibly) replicated across the intersection packet. As can be seen from Table 5.4, for the 4-ray packets the probability of having hit the same triangle is actually rather high.The second step uses a multi-pass approach for the shading of an i-set: All different surface shaders, which are referenced within the i-set, are sequentially executed. Each of these shading passes works on the full i-set, performing all shading computations for all 64 intersection points, while using bit masks for invalidation of non-related intersection points. In order to speed up the sequential scanning for different surface shaders, each surface shaders invalidates its shader reference in the i-set after execution, ensuring that the corresponding surface shader is not executed again. Table 5.4 shows also that for the test scenes only one to two shading passes per i-set are required. After accessing the material cache for a shading pass, no further cache accesses have to be performed, and the intersection points can be efficiently shaded in parallel using SIMD instructions. Note that the current implementation does not support software-based multithreading for the geometry or material caches.

For secondary rays, the same approach as Boulos et al. [BEL+07] is followed: to generate coherent secondary packets, each 8×8 primary packet generates one reflection packet (of up to 8×8 rays), multiple shadow packets (one per light source), etc. In order to simplify matters, the current implementation uses only a diffuse shading model with shadows, but without reflection or refraction rays.

5.7 Parallelization Across Multiple SPUs

So far, the only considered was to make ray tracing fast on a single SPE. However, each Cell has eight SPEs, and a dual processor system even has 16 of them. Since a homogeneous programming model is used, and therefore have no SPE-to-SPE communication, from a programmers perspective it makes no difference where the SPEs are physically located.

Keeping all 16 SPEs utilized requires efficient load balancing. A standard approach of defining the SPE's working tasks by subdividing the image plane into a set of image tiles can be used to accomplish an efficient load balancing. From this shared task queue, each SPE dynamically fetches a new tile, and renders it. As accesses to this task queue must be synchronized, the Cell's atomic lookup and update capabilities are employed: an integer variable specifying the ID of the next tile to be rendered is allocated in the PPE's main memory. This variable is visible among all SPEs, and each time an SPE queries the value of the variable, it performs anatomic fetch-and-increment. This atomic update mechanism allows the SPEs to work fully independently from both other SPEs and PPE, requiring no communication among those units. The only explicit synchronization is at the end of each frame, where the PPE waits to receive an *end frame* signal from each SPE. Figure 5.5 shows the efficiency of the dynamic load balancing for three test scenes, using the same settings as in the previous sections.

As image tile size 64×64 pixels are used resulting in 256 image tiles per frame. Even though the image tiles are only distributed across a single frame (which implies synchronization at the frame end), the approach provides almost linear scalability (without frame buffer transfer and shading) using

Scene	Same Triangle (4-Ray-Packet)	Passes Per Bundle	Cache Hits (in %) Vertices	Cache Hits (in %) Material
ERW6	97.18	1.005	99.54	99.99
Conference	88.74	1.23	96.11	98.04
VW Beetle	78.49	1.07	89.35	99.62

Table 5.4: Probability of an intersection packet (four rays) sharing the same triangle, and the cache hit rates for the vertex and material cache (direct mapped, 1 - 2 shading passes). Both, the vertex and shader cache have 64 entries; all scenes are rendered at 1024^2 (only primary rays).

up to 16 SPEs.

5.8 Overall Performance Comparison and Discussion

In order to evaluate the efficiency of this approach, it is compared to existing optimized alternative approaches. Figure 5.4 shows the used test scenes. On the Cell processor, no *real* alternative approach is published until now and other existing implementations follow exactly the above described strategies [MNM06], or at least a subset [FSY+06] (software caching), and therefore offer only similar or slower performance and are thus not intersecting for a comparison. Today's fastest published ray tracing results have all been realized on commodity CPUs (using either Pentium-IV CPUs [RSH05, WIK+06], or Opteron CPUs [WBS07]). To compare against these ray tracers, some of the scenes also used in [WBS07] and [WIK+06] are taken, and measured their performance. Table 5.5 reports the achieved fps on a single 3.2GHz SPE, as well as on a single and dual 3.2Ghz Cell processor evaluation system with 8 respectively 16 SPEs. As a baseline for comparisons, a x86-based implemen-

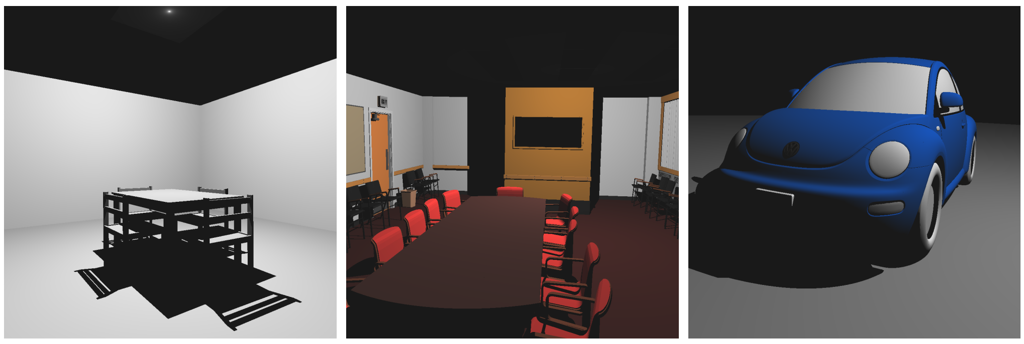

Figure 5.4: The test scenes: ERW6 with 804 Tris (left), the Conference-Room with 280K Tris (middle), and a VW-Beetle scene with 680K Tris (right).

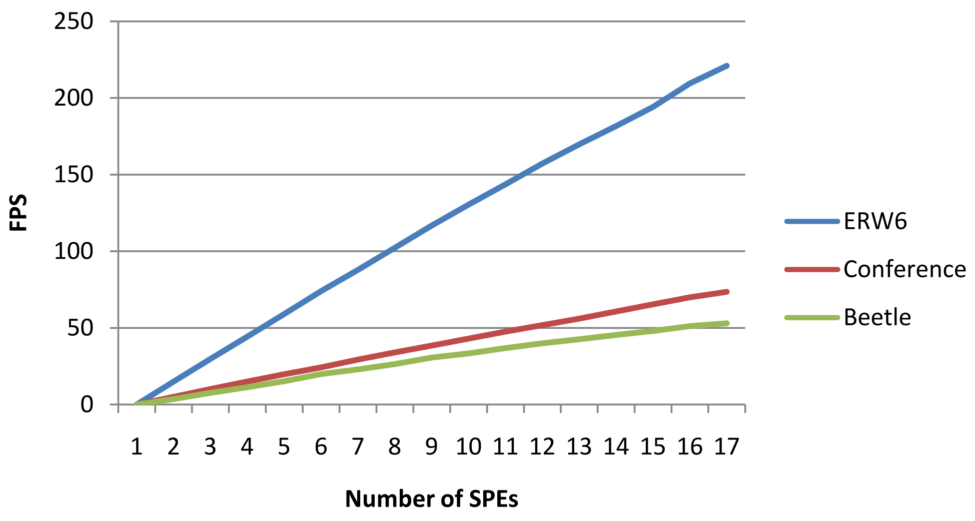

Figure 5.5: Scalability across several SPEs using dynamic load balancing based on image tiles. The atomic lookup and update capabilities of the Cell makes a fast and efficient implementation possible which is able to provide an almost linear scalability with up to sixteen 3.2Ghz SPEs. All tests were run with 256 BVH box cache entries and 256 triangle data cache entries.

tation of the algorithm proposed by Wald et al. [WBS07] is included; for a fairer comparison, Wald et al.'s code is not used, but a reimplementation that performs exactly the same intersection, traversal, and – in particular – shading computations as on the Cell, but with Opteron-optimized code (thus achieving roughly the same performance as Wald et al.'s system).

As shown in Table 5.5, the Cell-based implementation is quite efficient: on a single 3.2Ghz SPE, our implementation achieves a performance that is roughly on par with that achieved by one of the fastest known ray tracing implementations on a full-fledged Opteron CPU. As the system scales well over the available SPEs, the Cell system is then roughly twice as fast as a modern quad-core x86-based system.

Applying the profiling features of the Cell Simulator [Int05] allows for obtaining a detailed dependency stall analysis on the SPEs. As the current simulator does not provide a cycle accurate profiling of DMA transfers, cache accesses (and the related DMA transfers) are excluded from the analysis. For pure ray casting, roughly 25% of all cycles are taken by stalls: 11% for mispredicted branches, 10% by dependency stalls and 4% by branch-hit related stalls. The CPI ratio for complete BVH traversal is 0.99.

		2.4GHz	3.2 GHz Cells		
Scene	Shading	x86	1 SPE	8 SPEs	16 SPEs
ERW6	No	28.1	30.5	226.4	401.4
Conference	No	8.7	8.2	62.0	115.7
VW Beetle	No	7.7	9.2	61.1	109.4
ERW6	Diffuse	15.3	16.7	126.6	215.2
Conference	Diffuse	6.7	5.6	42.9	79.1
VW Beetle	Diffuse	6.0	4.4	36.1	65.3
ERW6	+Shadow	7.2	9.14	70.95	130.90
Conference	+Shadow	3.0	3.01	23.52	44.43
VW Beetle	+Shadow	2.5	1.89	15.21	29.78

Table 5.5: Performance in frames/sec on a 3.2GHz SPE, a single respectively dual 3.2Ghz Cell processor system, and a 2.4Ghz x86 AMD Opteron CPU using pure ray casting, shading, and shading with shadows (at 1024^2 pixels). For pure ray casting, the implementation on a single 3.2Ghz SPE is almost roughly on par with an up-to-data Opteron CPU. In addition, a Cell has 8 such SPEs, and can, in future, be clocked at a higher rate. Nevertheless, today, quad-core x86 CPUs are available and the Cell's performance advantage will degenerate quickly.

5.8.1 Shading and Secondary Rays

As can also be seen from Table 5.5, the Cell-based shading does not work as good as the traversal and intersection: in particular for more complex scenes in which rays hit different triangles, even simple shading becomes costly. For example, while for all test scenes pure ray casting on a single SPE is roughly as fast as on a AMD Opteron core, the Opteron is up to twice as fast once shading gets turned on. This, however, is not surprising, as this implementation is mostly focused on efficient ray traversal and intersection so far. In particular, profiling the shading code (using the simulator) shows that 43% of the cycles required for shading are wasted in stalls: 21.5% for dependency and 21.5% for mispredicted branches. This is mostly caused by inefficient instruction scheduling, so it can be expected that future compiler versions in combination with manual optimizations will provide a significant performance increase for the shading part. Secondary rays, on the other hand, do not further widen the performance gap between x86 and the Cell processor, as they again benefit from the optimized ray traversal kernel. In addition, shadow rays could be further accelerated using early shadow ray termination, which has not been applied, yet.

For highly recursive shading and realistic lighting effects it is still not clear

	Used Data in KB per Frame for				
Scene	BVH Nodes	TriAccels	Vertices	Materials	Total
ERW6	24	43	130	0.03	202
Conference	1,113	2,797	1,278	87	5,275
VW Beetle	3,724	4,766	4,303	153	12,794

Table 5.6: Required bandwidth to system memory for loading different types of scene data (in KB per frame). All scenes are rendered at 1024^2 pixels using simple shading, and with 256 entries for both BVH and triangle cache. Due to our caching framework, even the complex VW Beetle requires only a mere of 12 MB memory bandwidth per frame.

how to efficiently map them to a packet-based shading framework. First work has already been done [BEL+07], and it can be believed that most of this is directly applicable. However, the limitation of the Cell processor, e.g. limited local store size, makes the realization of complex shading even more challenging.

5.8.2 Caching, Software-Multithreading, and Bandwidth

As shown in the previous sections, caching works well for BVH nodes, but the cache hit rates for triangles quickly drop with an increasing scene complexity. Under the assumption that enough memory bandwidth can be reserved, one could abandon the triangle cache completely. However, for high frame rates the bandwidth could possibly limit the total performance.

Table 5.6 shows that for the complex VW Beetle scene only 12MB of bandwidth to memory is required. In particular, the largest part of the bandwidth is taken by loading triangle data (4.7 MB). Note that for writing the final color as 32bit RGB values to the frame buffer, 4,096 KB per frame of additional bandwidth is required. Even though a memory bandwidth of 12 MB per frame seems to be low, one should keep in mind that DMA transfers are not performed in large chunks of data, but with small granularities of 16, 32 or 64 bytes. Memory latency has therefore a much higher impact than memory bandwidth. For this kind of latency-bounded memory access, software multithreading is a useful approach.

5.9 Architectural Shortcomings

Even though the Cell – and, in particular, the SPEs – have a powerful architecture and instruction set, even small general extensions to the SPEs could further improve its efficiency for ray tracing e.g.:

Hardware Caching: In order to avoid redundant data loads from the PPE's main memory to the SPE's local store as much as possible, it is necessary to exploiting caching strategies (see Section 5.5). These caching strategies are costly in terms of clock cycles. In the currently used implementation a cache hit requires between 32 (direct mapped) and 44 (4-way) cycles. A cache miss more than 1000 (without multithreading). Even if 100% cache hits could be achieved every single data access costs still at least 32 cycles which significantly reduces the rendering performance. A future version of the Cell should, if possible, include a hardware caching mechanism to reduce this impact.

Hardware Multithreading: After exploiting software caching, it was consequent to implement software multithreading to reduce the impact of cache misses. Although it is possible to do that (see Section 5.5), it is rather complicated and each software engineer is forced to implement its own optimized version for specific problems, or to use (slow) libraries provided by the Cell-SDK. Hardware support for multithreading, e.g. two threads, would speed up all kinds of applications with complex and poorly predictable memory access patterns, and relief developers from this task.

Larger local store Size: Currently, the size of the local store per SPE is limited to 256KB memory. For some applications this may not be sufficient, e.g. a shader library with many complex shaders and preprocessed data might easily exceed this size. Another problem is that it is common to use templates, e.g. in C++, to generate at compile time different optimized execution path for several functions like the BVH traversal for ray bundles with a common origin and bundles that do not. This can also lead to too large programs for an SPE and care has to be taken to stay below the available memory limit. Although it is possible to page in and out functions on the fly, for realtime applications this is not an option.

DMA Setup: The shading part requires many data gather operations, e.g. loading four word elements from four different locations in the local store, where the four addresses are held within a single register. As each of these word elements does not need to be aligned on a sixteen byte boundary, a long and costly instruction sequence (scalar loading) is required to load and arrange the data. Therefore, an extended load instruction would be very helpful.

Hardware Transcendental Functions: In ray tracing, transcendental functions can be used for shading computations or to generate new secondary rays e.g. uniformly distributed rays over a hemisphere in a Monte Carlo-based rendering algorithm. Unfortunately, there is no hardware support and software functions have to be used. If it is considered that millions of transcendental function have to be computed in a global illumination simulation per frame the required 74 cycles, e.g. for a `sin`, are quite costly.

Extended Branching: Only a single branch hint can be specified at a time (fixed at compile time), and this must be placed at a certain distance before the branch. This results in an increased number of branch mispredictions for branch intense code. Specifying branch hints for multiple branches in advance, and dynamic branching where the branch target can be changed during runtime could significantly reduce the misprediction rate.

Although these suggested changes require significant modifications to the Cells circuit design, they have the potential to speed up all kinds of algorithms including ray tracing.

5.10 Conclusions and Future Work

In this chapter, it is shown how ray tracing can be efficiently mapped to the Cell architecture. The resulting ray tracing performance of the presented approach is on a single SPE roughly comparable to the fastest known x86-based systems (running on a single core). Using a SIMD enabled BVH traversal and optimized routines, a per-SPE performance of several million rays per second can be achieved. Access to memory is handled via explicitly caching scene data, and software-multithreading is used to bridge cache miss latencies. In addition, a load-balanced parallelization scheme achieves nearly linear scalability across multiple SPEs, thereby using all of the Cell's computational resources. The remaining bottleneck is shading, which requires many cache accesses, costly data gather operations, and a complex control flow, making the Cell architecture less efficient than a commodity x86 core. Therefore, future modifications should directly concentrate on a simplified but efficient shading framework e.g. using a compiler approach. For example, a shading compiler for a RenderMen like language could hide all ray bundle, threading, and caching issues from a programmer to simplify the development of new advanced shaders. On the software side, a full-fledged ray tracing system should also integrate multi-threading in the shading stage. As the SPEs are exclusively designed for high clock rates, it can be expected that future versions of the Cell processor to have a higher clock rate and an increased number of SPEs. Even the current generation of SPEs has been reported to run stable at 5.2 GHz [ADF$^+$] by Asano et al.; so a great performance boost from future generations can be expected. But still, the goal to achieve suitable ray tracing performance for future computer games is not fulfilled and future revisions of the Cell processor, possibly including the proposed modifications in Section 5.9, have to be evaluated from scratch again.

Part II

Isosurface Visualization using Ray Tracing

Chapter 6

An Introduction to Isosurface Rendering

The first part of this thesis is focused on ray tracing of (dynamic) surface-meshes and its relation to future computer games and hardware platforms. In this second part, the focus is on *volumetric datasets* in the context of scientific visualization. More precisely on the subproblem of isosurface rendering. Due to the fact that volume rendering uses fundamentally different primitives and rendering methods this introduction outlines the most important concepts and algorithms.

6.1 Volume Rendering and Volumetric Data

Volume rendering is a method to generate 2D images from 3D volumes [HJ04]. The goal is to visualize the volumetric data such that important properties can be intuitively examined and inner, otherwise hidden, structures are exposed [SM00].

Volumetric datasets are used in many application areas such as medicine [WR02], forensic [MMU05] and fluid mechanics [MCC+99] to name just a few. Primary sources for volumes are CT (Computed Tomography) or MRI (Magnetic Resonance Imaging) machines or numerical simulations. In essence volumetric datasets consist of scalar values [SM00] defined at discrete spatial locations. For this reason, the volumes are also referred to as scalar fields. Each of those scalar values – called *voxel* for volumetric element – represents a physical quantity like pressure or density and are the smallest element of computation in volume rendering. A voxel can be seen as the 3D equivalent to a 2D pixel (picture element). The data types used for the voxels can be arbitrary ranging from simple binary values up to floating point numbers. Nevertheless, it is common that all voxels have the same data type in a dataset. Typically, scanned datasets have a voxel quantization of eight, twelve, or 16 bit and represent integers whereas numerical simulations generate often 32 or 64 bit floating point numbers.

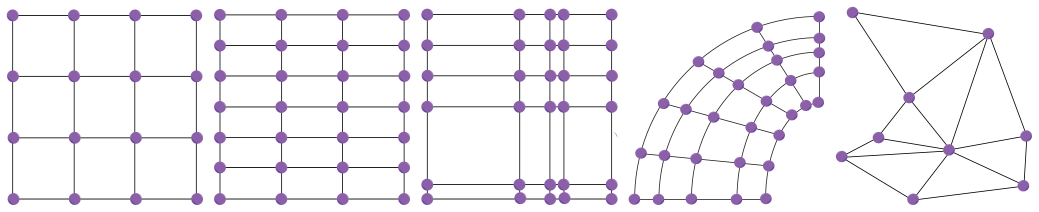

Figure 6.1: 2D examples of various volumetric grid types (from left to right): regular, anisotropic regular, rectilinear, curvilinear, and unstructured.

The structure of these scalar fields is diverse. Principally one can distinguish between *structured* and *unstructured* datasets. Structured datasets have an inherent organization that allows for simple addressing computations of voxels in a grid. By contrast, unstructured datasets require additional information about the topology of the grid in the form of an adjacency list, i.e. to determine the neighbors of a voxel. Figure 6.1 exemplifies some widely used volumetric grid types and their terminology.

For regular and rectilinear datasets *cells* can be defined by eight grid points (see Figure 6.2 left). Curvilinear datasets can also use this cell concept by transforming the volume from its *real* physical space to a *computational space* [WCA$^+$90] that maps the twisted grid into a regular one. In the case of unstructured grids the voxel-soup is usually partitioned, e.g. into a tetrahedral mesh, for further processing. Whenever in this thesis unstructured grids are used, they are converted first to such tetrahedral meshes. Each tetrahedron can then be considered as a cell. In the following the eight voxels of a cubic cell are addressed by $c_{i,j,k}$ with $i, j, k \in \{0, 1\}$. The voxels of a tetrahedron by t_i with $i \in \{0..3\}$.

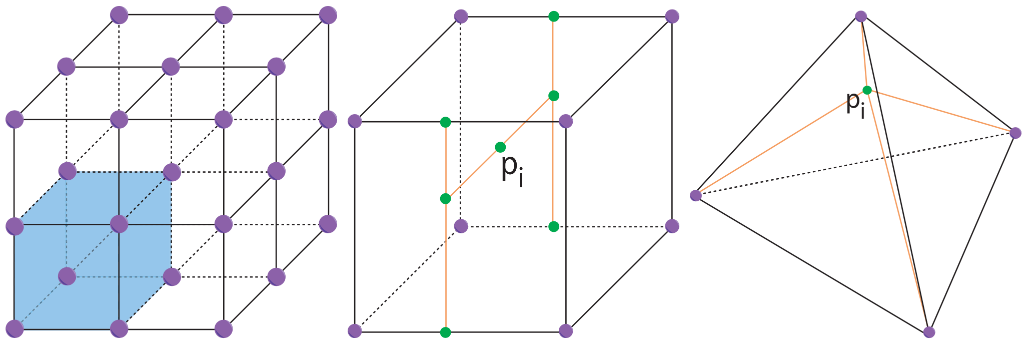

Figure 6.2: Left: a cell (light blue box) defined by eight voxel locations. Middle: within a cubic cell at each location p_i a value can be reconstructed by trilinear interpolation. Right: For tetrahedra a linear interpolation can be used to reconstruct in-between values using barycentric coordinates.

6.1.1 Reconstruction and Interpolation

Since a volume is only defined at some discrete locations in space, it is necessary to reconstruct in-between values.

Cubic Cells: In the case of regular, rectilinear and curvilinear datasets, a trilinear interpolation is usually applied within a cell (see Figure 6.2 middle) but higher, and lower, order interpolations are also possible [The01, RZNS04]. The computations to trilinearly interpolate a scalar at the local cell coordinates (x, y, z) are shown in Equation 6.1:

$$f(x, y, z) = \sum_{i,j,k \in \{0,1\}} x_i y_i z_i v_{i,j,k} \tag{6.1}$$

$x_0 = 1 - x$ and $x_1 = x$, $y_0 = 1 - y$ and $y_1 = y$, and $z_0 = 1 - z$ and $z_1 = z$.

Tetrahedral Cells: For unstructured datasets it is common to exploit a linear interpolation scheme within the tetrahedra. Figure 6.2 shows the correlation between cells and their commonly used interpolation methods.

Given a point $p_i = (x, y, z)$ in an *arbitrary* defined cell with n voxels, a scalar $f(p_i)$ can be reconstructed as a weighted sum of the n voxels v_i:

$$f(p_i) = \sum_{i=0}^{n-1} f(t_i) w_i; \tag{6.2}$$

The weights w_i are then the barycentric coordinates of p_i and can be computed e.g. for a tetrahedron by solving the equation system $p_i = \alpha A + \beta B + \gamma C + \delta D$ for $\alpha, \beta, \gamma, \delta$ with A, B, C, and D being the coordinates of the tetrahedrons vertices; and $\alpha + \beta + \gamma + \delta = 1$.

6.2 Volume Visualization Techniques

Typically the literature differentiates between five volume rendering techniques, each with a special field of application [SM00, LCN98, HJ04]. See Figure 6.3 for typical images that are generated by these methods.

Decomposition: Decomposition methods show the dataset on a per voxel or slice basis. In a voxel based approach, for every voxel a geometric representative is generated, e.g. a sphere, that is scaled and/or colored according to its data value at the voxel. A spacing between the primitives allows to view inside the volume – at least to a certain degree. Slices are commonly, but not necessarily, axis aligned layers of the volume. They can be rendered in many ways like (colored) textures, height maps, or be used for a further processing like segmentations of certain regions [NH90].

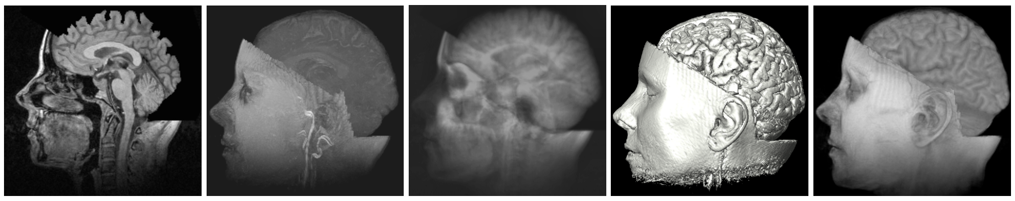

Figure 6.3: An MRI scanned head dataset rendered using various volume rendering methods. a) A typical slice view. b) Maximum Intensity Projection. c) X-Ray rendering. d) An isosurface. e) The head rendered using semi-transparent volume rendering. Please note how each of these methods let a user interpret the dataset in a completely different way.

Isosurfaces: In some applications it is important to investigate the distribution of a certain single value (the *isovalue*) within a dataset. In particular it is sometimes interesting to examine the topological and geometrical properties of these distributions. The visualization of all points within the dataset with the same isovalue (aka. as level set) yields in most cases a surface – or several – and therefore this method is called isosurface rendering. An isosurface can formally be defined as the set of all points within the volume that fulfill $f(x, y, z) = c$. c is then a user specified isovalue. An important application for isosurface rendering is e.g. virtual endoscopy.

Maximum Intensity Projection (MIP): For each pixel the MIP method computes the maximum value that can be encountered along a ray that pierces a volume. This method is often used for Magnetic Resonance Angiograms where thin structures, e.g. blood vessels, have to be rendered accurately and other visualization methods fail to preserve such fine details. Unfortunately MIP rendered images do not comprise a good visual sensation of depth such that it cannot be decided from a still image what is in front or back. Due to that, MIP applications commonly rotate the view around the object to increases the depth perception.

Physically based Models: Physically based volume rendering models consider the volume as a transparent medium. If light passes this medium it can be *absorbed*, *scattered*, or initiate new light *emission*s. These effects are captured with the general Radiative Transfer Equation [SH92, Max95, WMS98]. A transfer function [Sab88, Lev88, UK88] can be used to map the voxel quantities to optical properties. These semi-transparent rendering models are in particular useful when the whole dataset should be visualized at once and individual surfaces are not (that) important.

X-Ray: X-Ray rendering is a special case of the physical based models which only considers radiance absorption [DCH88]. Metaphorically speaking a ray that pierces a volume integrates the absorption along its way through the volume. This results in typical x-ray machine pictures and can thus be used for the same purpose.

6.3 Volume Ray Tracing

After having described various techniques to visualize volumetric datasets, it is interesting to note that all of these methods can be rendered with a unified volume ray tracing approach, e.g. only a single index structure is required for isosurface, MIP, and semi-transparent rendering. Volume ray tracing works fundamentally in the same manner as surface ray tracing except its underlying primitive – the volume and its cells – needs to be treated differently. In fact a volume is just another render primitive for a sophisticated ray tracing engine.

In volume ray tracing a ray does not just hit a volume like a surface at a single point, but *spans* the volume and pierces some of the its cells. The task of volume ray tracing is to enumerate all *relevant* pierced cells and to perform a *special operation* on each of them. Which cells are relevant and what kind of operation per cell is performed depends on the volume rendering method. For example, in the case of MIP, for each relevant ray segment that spans a cell the maximum intensity has to be computed. Each cell that potentially bears a higher intensity along its ray segment has to be checked. Physically based models integrate the radiance of each ray segment. All cells that have a zero contribution are not relevant and should be skipped for performance reasons. Isosurface rendering seeks in the relevant cells the first hit point of the isosurface and the ray. In isosurface rendering a cell is relevant if potentially a ray isosurface hit point can be found.

Again, as in the case of surface ray tracing, realtime implementations have to reduce the number of cell operations that do not contribute to the final result, and the costs per visited cell. A thorough discussion can be found in [MFS06]. As isosurface rendering is the focus of this second part of the thesis, Section 6.5 will discuss it in more detail.

6.4 Normal Estimation

For some volume rendering techniques, in particular for isosurfaces, it is useful to perform shading computations. However, a volume is just a ”point cloud” and no shading normals are a priori available – but required even for simple shading models. Therefore it is necessary to *estimate* normals at shading locations. A common method is to compute a *gradient G* using *finite differences.* The gradient is, in this context, an unnormalized vector that points away from regions of high quantities and is perpendicular to the isosurface. A normal N can then be estimated by $N \approx G/|G|$. Among the finite difference methods the *central difference* is commonly used because it offers the best trade-off between computational costs, memory accesses, and estimated gradient quality. Equation 6.3 shows the calculations:

$$\begin{aligned} G_x &= \frac{f(x-\Delta_x, y, z) - f(x+\Delta_x, y, z)}{2\Delta_x} \\ G_y &= \frac{f(x, y-\Delta_y, z) - f(x, y+\Delta_y, z)}{2\Delta_y} \\ G_y &= \frac{f(x, y, z-\Delta_z) - f(x, y, z+\Delta_z)}{2\Delta_z} \end{aligned} \tag{6.3}$$

Given a global sample location at (x, y, z), six values have to be reconstructed for the gradient estimation. The distances Δ_i are best chosen such that interpolation locations in neighboring cells have the same local coordinates as (x, y, z) – assuming anisotropic grids. Another possibility is to calculate first the central differences at a cells voxel locations, and then to interpolate a gradient at (x, y, z) using these corner gradients. Although this seems to be computational more expensive this *derivative first* approach [MMMY97] may pay-off when those corner gradients are cached and reused [GBKG04]. For most applications the central difference is a sufficient gradient estimation scheme but produces artifacts when close-up views are rendered [Gri05]. A thorough discussion of other normal estimation methods and their computation-quality trade-off can be found in [MMMY97, YCK92].

6.5 Isosurface Rendering

In order to render isosurfaces basically two different approaches exists: *extraction* and *direct rendering*. The first approach extracts *prior* to rendering a polygonal approximation of the isosurface that is afterwards passed to a rendering system. The latter one uses e.g. ray tracing to directly render the isosurface without the extraction preprocess. The next two sections describe both methods in more detail.

6.5.1 Isosurface Extraction

Probably the very first approach to render isosurfaces was presented by Keppel [Kep75] in 1975. Keppel divided the isosurface generation into two steps. In the first step *isolines* were computed on each 2D slice of the volume, i.e. in the x-y plane. Afterwards these isolines were connected by triangles. Unfortunately the isoline connecting algorithm is far from trivial as it is not always clear which contours belong together and user interaction is sometimes required to resolve ambiguous cases. Automatic approaches do exist today but do not guarantee a correct result [SM00].

A simpler approach, that does also not require any user interaction, to display isosurfaces was presented by Hermann et al. [HL79] in 1979. Their idea was to consider a voxel as a homogeneous box, called cuberille with a certain extend. For each voxel that matches the isovalue simply a six sided box is rendered resulting in *blocky* approximations of the isosurface. However, the overall computational costs for extracting and rendering an isosurface were reduced significantly. Furthermore, this algorithm is very simple and does not produce any ambiguities.

However, today isosurface rendering is dominated by extraction algorithms, like Lorensen et al's. *marching cubes* approach, that work directly on the (cubic) cells of a (rectilinear) grid. Marching cubes creates a polygonal approximation, i.e. with triangles, directly for each cell that contains a part of the isosurface. In the following a cell that contains a part of the isosurface is referred to as *boundary cell*. To determine whether a cell is a boundary cell, the isovalue is compared to the cells eight voxels. If the voxel value is greater than the isovalue it is marked with a $+$, and with a $-$ when it is less. When all marks of a cell are equal, the isosurface does not cross this cell and no further computations are necessary. In order to generate polygons in the boundary cells, first some vertices have to be computed. On each cell's edge with different marks, a vertex is generated at a location that can be computed by linear interpolation. Finally the generated vertices can be triangulated. The same approach can also be used for other kinds of cells like tetrahedra [DK91a].

After the initial development of the marching cubes algorithm, it has been extended in many forms: e.g. in the better handling of topological ambiguities that can appear in the triangulation step [NH91], higher efficiency for larger data sets [CMM+97, LSJ96], view-dependent extraction methods [LH98], and adaptive or multi-resolution methods [WKE99] to name just a few.

6.5.2 Isosurface Ray Tracing

An alternative to isosurface extraction is to *directly* compute the isosurface, either by some form of preintegrated direct volume rendering [RSEB+00, EKE01] or by ray tracing, i.e., by computing the intersection of rays with the implicit function $f(x, y, z) = c$. Due to the high computational cost, realtime isosurface ray tracing was first realized on supercomputers by Parker et al. [PSL+98, PMS+99] but is nowadays also feasible on modern GPUs as well [WS01].

Isosurface ray tracing consists of the same two task as surface ray tracing:

a (front-to-back) traversal scheme that enumerates the boundary cells along the ray, and an intersection test between a ray and the isosurface. A brute force approach would use e.g. ray marching, like [AW87], to traverse through the volumetric grid, check if a cell is a boundary cell, and if that's the case perform an intersection test. This procedure repeats until the first hit point is found. In Chapter 7 a novel memory-efficient index-structure based on kD-trees is described to speed up the boundary cell enumeration in rectilinear grids. Chapter 8 shows how to use BVHs for this task with unstructured volumes.

The ray isosurface intersection test can be performed in two ways. Either some polygons are generated to approximate the isosurface in a cell, i.e. via marching cubes[1], and the intersection is computed with these polygons (see Section 2.4.4) or the intersection is performed analytically with the ray and the isosurface.

For example in cubic cells, an analytic solution can be derived by substituting the ray equation $\mathcal{R}(t) = \mathcal{O} + t\vec{\mathcal{D}}$ into the trilinear interpolation formula (see Equation 6.1) which yields:

$$f(t) = \sum_{i,j,k \in \{0,1\}} (o_x + td_x)(o_y + td_y)(o_z + td_z)c_{i,j,k} \tag{6.4}$$

When this sum is expanded and solved for the distance parameter t a cubic polynomial is obtained that can be solved e.g. using Cardanos formula [MFK+04].

Compared to extracting the complete explicit tessellation of the isosurface in the first place before the rendering starts, direct ray tracing has several advantages. First of all, ray tracing directly supports global effects and scales well with the scene size (as discussed in Chapter 2 and 3). Second, ray tracing does not rely on a polygonal approximation of the volume function, as it is possible to compute the intersection analytically. Even in cases where a polygonal approximation is preferable, using ray tracing it is possible to extract only those polygons that are visible since the algorithm is absolutely view dependent. Finally, since their is no explicit tessellation, no new approximation has to be generated when the isovalue is changed. This allows then to change the isovalue arbitrary at any time.

[1]In this case ray tracing is used to extract the geometric approximation in a view dependent manner.

Chapter 7

Isosurface Ray Tracing of Rectilinear Volumes

This chapter describes a novel memory-efficient data structure based on kD-trees for interactive isosurface ray tracing of rectilinear volumes, including an extension for out-of-core data sets. Furthermore a new fast and accurate ray-isosurface intersection test is proposed. The overall approach allows to inspect even massive (out-of-core) data sets in the gigabyte range at interactive frame rates including on-thy-fly isovalue changes.

7.1 Isosurface Ray Tracing using Implicit kD-trees

kD-trees and BVHs are well-known for representing polygonal scenes, where they often outperform other data structures, as they can adapt much better to the scene's geometry [Hav01b, Wal04]. This is particularly the case for scenes with highly varying primitive density, as these usually contain large regions of empty space that a well-built kD-tree or BVH can traverse with very few traversal steps.

For highly regular scenes such as a 3D volume however these advantages – varying geometry density and empty space – cannot be exploited, and the large amount of inner nodes in a kD-tree is usually detrimental in both memory overhead and number of traversal steps. Thus, grid-like data structures are usually better traversed from cell to cell with a voxel walking algorithm such as the one by Amanatides et al. [AW87] or Wald et al's. [WIK$^+$06] coherent grid traverser. For example, just finding the starting voxel of a ray incurs logarithmic cost for a kD-tree, while in a grid it can be found in constant time.

While these arguments are undoubtedly true for rectilinear (volume) datasets, they do not necessarily hold for the specific task of rendering isosurfaces defined by such a volume data set: While the set of data points in fact **is** a regular 3D grid, the isosurface defined by that data set is only located within the small subset of boundary cells. These boundary cells share again

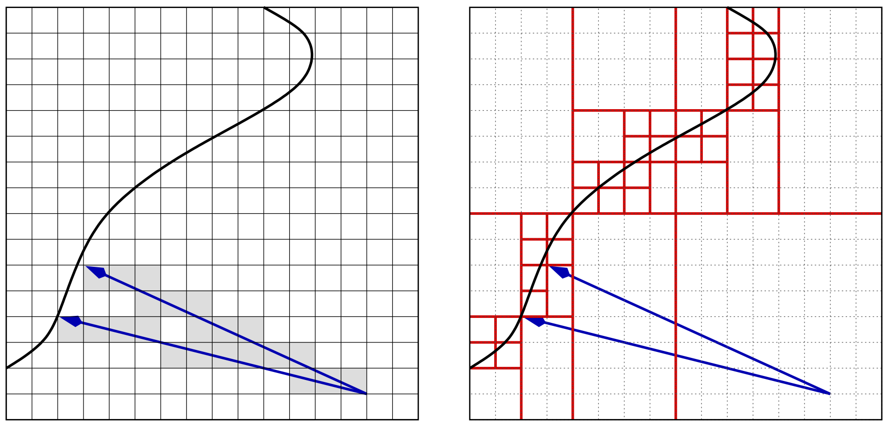
Figure 7.1: The small, regular cells in a regular volume data set offer few potential for exploiting the advantages of kD-trees and packet traversal, as rays have to perform many traversal steps, and diverge quickly. However, considering only the boundary cells of an isovalue, a kD-tree allows for quickly skipping large regions of space.

many properties with primitives in polygonal ray tracing: They are irregularly distributed, sparse, and often enclosed by large regions of *empty space*. This once again is the environment for which hierarchical index structures are ideally suited (also see Figure 7.1). As a BVH plays to its strength only when geometry is heavily overlapping, or changes (due to its cheap update capabilities), what is not the case here, a kD-tree is chosen as basic data structure.

7.1.1 The Implicit Min/Max kD-tree

This only leaves the question how to best use a kD-tree for representing the isosurface. If one were willing to restrict oneself to rendering a fixed isovalue only, one could identify all boundary cells in advance, build a kD-tree only over those "primitives", and expect to reap all the benefits of kD-trees also for isosurface ray tracing. Unfortunately this would no longer allow for interactively changing the isovalue. Instead, a kD-tree that contains all possible isosurfaces at the same time is build. Each kD-tree node is annotated with information on what isosurfaces it contains, and performs the classification *implicitly* during traversal. Knowing which isovalues are contained within a subtree allows to easily skip entire subtrees of cells that do not contain the required isovalue, and thus – implicitly – traverse a kD-tree only containing boundary cells of the current isovalue.

In order to realize this implicit kD-tree, only two ingredients are required: First, a kD-tree in which each node maintains information on what isosurfaces

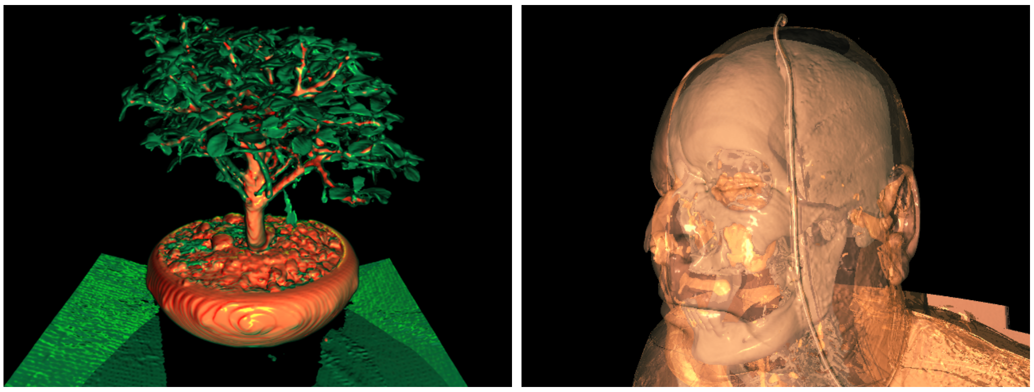

Figure 7.2: Implicitly culling non-contributing branches of the implicit kD-tree during traversal also allows for rendering multiple isosurfaces at the same time. Left: The bonsai tree, with a green isosurface for the leaves, and a brown one for the trunk. Right: The Visible Female's head, with isosurfaces for bones and semitransparent skin.

are contained within its subtree. Second, a modified traversal algorithm that traverses a kD-tree, but implicitly classifies each visited node for whether it can actually contain the queried isovalue and skips it if that is not the case. As the tree still encodes the whole data set, the isovalue can still be changed on-the-fly. As a side effect, this approach also allows for searching for several different isosurfaces concurrently within the same traversal operation, as one can trivially base the culling operation on multiple isovalues at the same time. This is particularly important for scenes in which multiple isosurfaces – e.g. both skin and bone – are of equal interest (see Figure 7.2).

7.1.2 Building the Implicit Min/Max kD-tree

Building the implicit min/max kD-tree consists of two interleaved steps. First, a kD-tree over all the cells of the entire data set is recursively built until a leaf contains a single cell. This is done in a way that a kD-tree split plane always coincides with the cell boundaries of the volume cells, yielding a one-to-one mapping between the volume's cells and the voxels of the kD-tree. Currently, the volume is split in the middle at the cell boundary that is closest in the center of the largest dimension. Second, each leaf node stores the min/max values of its voxels, and each inner node stores the min/max values of its children in order to annotate the range of isosurfaces contained within a subtree. Note that this data structure is similar to the one used by Wilhelm and van Gelder [WV], except that a kD-tree instead of an octree is used and that the kD-tree is used for efficient ray traversal instead of for isosurface extraction.

7.2 Efficient Traversal and Intersection

As mentioned before this isosurface rendering system is to be integrated into the current OpenRT system. In the current state, OpenRT only supports single rays and four-ray packets which means that here no larger ray bundle algorithms are exploited – but could if necessary. In the next two sections ray traversal, closely following [Wal04], and various ray-isosurface intersection tests, including a novel one, are discussed.

7.2.1 Ray Traversal

The data structure for each kD-tree node is – except for the min/max values stored per node – very similar to the polygonal case (outlined in Section 2.4.3). Thus, the already well-known polygonal traversal code requires only minimal modifications: During each traversal step first it is tested whether the current isovalue lies in the min/max range specified by the current node. If this is not the case, this subtree is immediately culled, and the far node is processed. Otherwise, exactly the same operations as in the polygonal case (see Section 2.4.3 and [Wal04, Hav01b, Ben06] for more details) are performed. The culling can be realized by two simple compares $f_{iso} \geq f_{min}(node)$ and $f_{iso} \leq f_{max}(node)$ in one additional conditional in each traversal step. Although these tests have to be performed in every traversal step, they are still quite affordable (see below).

As already discussed in Section 2.4.1, the efficiency of a SIMD packet traversal code depends to a large degree on the average utilization of the SIMD units, i.e. on the average number of rays that are active in a packet. As the cells of a volume data set are often quite small in comparison to the screen resolution one could naïvely expect the SIMD efficiency to be small as well, as different rays may traverse different nodes. When using a kD-tree however, for most of the traversal steps in the upper tree levels the rays stay together (see Table 7.1). As expected, the SIMD traversal suffers from a lack of coherence for distant views of high-resolution data sets, in particular for low screen resolutions. For less extreme settings however, the SIMD code allows for reducing the number of traversal steps by up to a factor of four, and in practice works quite well. Nonetheless, for high resolution data sets and large ray bundles, as used in Section 5.3 for BVHs, the coherence drops to an unusable extent resulting in single ray performance [KWPH06].

7.2.2 Isosurface Intersection

While traversing the kD-tree data structure is almost the same as in polygonal ray tracing, when reaching a leaf the situation changes. While in polygonal ray tracing a leaf contains a list of triangle IDs, here the tree is built such that each leaf contains exactly one "primitive" – a single cell. As described in

Section 6.5.2 computing an exact ray-isosurface intersection requires to setup a cubic polynomial in order to find the smallest root that lies in the interval $[t_{near}, t_{far}]$, the ray segment overlapping the cell. As this ray-isosurface intersection is considerably more costly than a ray-triangle intersection special care has to be taken to implement this operation efficiently. In the following, the existing *fast but approximative*, and *exact but slow* intersection tests will be discussed in order to subsequentially motivate a novel **fast and exact** intersection method.

Approximative Intersection Tests

Approximative intersection tests consider the function along the ray not as cubic but mostly linear. For polynomials with two roots these tests are likely to fail and for three roots a wrong or no solution could be calculated. In practice however, the artifacts that can be introduced, i.e. by not finding a hit point, are quite tolerable [NMHW02, MFK$^+$04] and cannot be observed very often, if at all, since it is very rarely the case that a cell contains multiple, unconnected, and overlapping isosurface parts (which also depends on the viewing direction) that would result in a full cubic polynomial.

Linear Interpolation: Probably the most simple way of performing an intersection test is to just assume that the function along the ray is linear. For the complete intersection test then three steps are necessary: First, compute $f(t_0 = t_{near})$ and $f(t_1 = t_{far})$ by trilinear interpolation [1]. Then, iff the isovalue c is $f(t_0) < c < f(t_1)$ the ray hits the isosurface. Finally, the hit distance t within the cell can be computed by $t = (c - f(t_0))/(-f(t_0) + f(t_1))$.

Neubauer-Iteration: One interesting extension to the linear intersection test is Neubauer's approximate intersection [NMHW02] method, which computes the intersection distance iteratively by repeated interval subdivision to refine the hit distance accuracy: First, $f(t_0 = t_{near})$ and $f(t_1 = t_{far})$ are computed by trilinear interpolation as above. If for the isovalue c, $f(t_0) < c < f(t_1)$ is true, an additional value $f(t_c)$ for a new interval $[t_0 < t_c < t_1]$ is computed, again using trilinear interpolation. Depending on $f(t_c)$, one then selects either the interval $[t_0, t_c]$, or the interval $[t_c, t_1]$, and iterates. Typically two or three iterations are chosen to refine the hit distance. Although this intersection test lacks the same problems as using a single linear interpolation the rendered isosurfaces have a somewhat "smoother" appearance.

[1] Although only bilinear interpolations are necessary it turned out that a trilinear interpolation is cheaper because this avoids to determine the face of the cell for which the bilinear interpolation has to be performed.

Exact Intersection Tests

Not just relying on a linear function but the analytically derived cubic polynomial $f(t) = at^3 + bt^2 + ct + d$ (usually reparameterized to $t \in [0, 1]$ for numerical stability) finds all three possible roots (intersection points). Nevertheless, in a basic approach this method requires finding all three potential roots before being able to determine the smallest valid one. One of the best-know methods to explicitly compute all roots is Schwarze's cubic root solver [Gla90] and is used e.g. in [PSL+98, DPH+03] and [Shi02][2].

Schwarzes Cubic Root Solver: After the coefficients a, b, c are calculated the degree of the polynomial is determined. Then, dependent on the degree an appropriate root solver is used i.e. the p, q-formula for quadrics. For *real* cubics, $a, b, c \neq 0$, a variant of Cardano's approach is used. Unfortunately, this requires costly trigonometric and hyperbolic functions and is numerically problematic, in particular when using single-precision floats e.g. for divisions.

An Exact Improved Iteration Method: Until now, the intersection test where either fast but approximative, or slow and accurate. By combining the ideas of Neubauers and Schwarzes approach a fast **and** accurate intersection test can be derived. Rather than computing all roots of the polynomial, an alternative is to use an iterative method with a proper start value. For finding the start value, the three intervals defined by $[-\infty, +\infty]$, $[t_{near}, t_{far}]$, $[t_0, t_1]$ have to be considered, where t_0, t_1 are the extrema of $f(t)$ (computed by solving $f'(t) = 3at^2 + 2bt + c$ using the p, q-formula).

From these three intervals, the first interval $[a, b]$ with $t_{near} \leq a$, $b \leq t_{far}$, and $sign(f(a)) \neq sign(f(b))$ contains only one root, which is the one that is searched for. This root can then be found by midpoint interpolation, i.e. by computing the midpoint $c = 0.5 * (a + b)$, then choosing the next appropriate interval either $[a, c]$ or $[c, b]$, and iterate. Except for one square root (for computing the extrema), this method uses only adds and muls, thus can be well implemented with SIMD, and is numerically very stable.

Like Schwarze, the improved iteration requires to explicitly compute the coefficients of the polynomial. Though this carries some initialization cost, knowing the coefficients also allows for computing $f(t)$ by evaluating the polynomial, which is much faster than through the trilinear interpolations performed by the Neubauer method. Thus, the improved iteration method is similarly fast as Neubauer iteration, while being exact in all cases.

```
f0=f(tmin); f1=f(tmax); //interpolate f0, f1
if(hasRealRoots(A,B,C,D) == true) {
```

[2]Please note that the term *exact* does not refer to the numerical precision of the solution but to the fact that all roots of the polynomial are found.

```
    e0=getFirstRoot(A,B,C,D);
    if(inRange(e0, t0, t1) == true) {
        if(sign(f(e0)) == sign(f0)) {
            t0=e0; f0=f(e0) //second segment
        }
        else {
            t1=e0; f1=f(e0);
        }
    }
    e1=getSecondRoot(A,B,C,D);
    if(inRange(e1, t0, t1) == true) {
        if(sign(f(e1)) == sign(f0)) {
            t0=e1; f0=f(e1) //third segment
        }
        else {
            t1=e1; f1=f(e1);
        }
    }
}
if(sign(f0) == sign(f1)
    return false;
for(int i = 0; i < N; i++)
{
    t=t0+(t1-t0)*(-f0/(f1-f0));
    if(sign(f(t) == sign(f0)) {
        t0=t; f0=f(t);
    }
    else {
        t1=t; f1=f(t);
    }
}
thit=t0+(t1-t0)*(-f0/(f1-f0));
```

Listing 7.1: Pseudo-code for the new improved intersection method

SIMD Considerations

After SIMD traversal allows for significantly reducing the number of traversal steps, it would be highly beneficial to use a SIMD variant for isosurface intersections as well. Due to the high computational density of the ray-voxel intersection, a SIMD variant that intersects a packet of four rays in parallel can be implemented quite efficiently, and achieves good speedups (see Table 7.1). Nonetheless, in the beginning there were two (related) issues that made the SIMD isosurface intersection problematic.

		512 × 512			1024 × 1024		
Data Set	Resolution	C	SIMD	Ratio	C	SIMD	Ratio
Aneurism	256^3	307	186	1.65	1229	545	2.25
Bonsai	256^3	544	360	1.51	2184	1027	2.12
ML	32^3	215	79	3.20	3451	927	3.55
ML	128^3	786	381	2.06	786	381	2.06
ML	512^3	680	646	1.05	2718	1863	1.46
Female	$512^2 * 1734$	179	177	1.01	716	708	1.01
" (zoom)	$512^2 * 1734$	384	98	3.99	1535	390	3.99
LLNL	$2048^2 * 1920$	631	632	0.98	2523	2520	1.02
" (zoom)	$2048^2 * 1920$	907	228	3.98	3630	909	3.99

Table 7.1: Number of surface intersection tests (in thousands) for both single-ray and SIMD traversal code, for various scenes and screen resolutions. The ratio column reveals the average number of active rays in a 4-ray packet. Due to the loss of coherency at the leaves, for extreme settings the number of intersections can even be slightly *higher* than in the single-ray code. For less extreme setting however the SIMD code can still achieve reasonable reductions in the the number of voxel intersections computed.

First, the SIMD efficiency for isosurface intersection is usually much lower than for packet traversal, as – in contrast to traversal – an intersection is always performed at the level where the rays are most incoherent. Therefore, intersection coherence is usually much lower than the average traversal coherence, and often very few rays are actually still active in a ray packet when reaching the leaf cells (see Table 7.1). Unfortunately SIMD code often bears some overhead as compared to a single-ray variant, which only pays-off if the SIMD code can be used for many rays in parallel. If however only a single ray is active, any potential overhead of the SIMD implementation may even result in a reduction of the overall performance.

Second, upon successful intersection a large number of values have to be stored to update the current information: hit flag, hit distance, local cell coordinates, and shadig normal – for a total of eight values per ray (128 bytes total). As these values are only stored for those rays that actually had an intersection, each of these stores in SIMD mode has to be realized with several masking operations to implement *conditional moves* [Adv03]. This turnes out to consume a significant portion of compute time, and leads to a significant overhead, which – combined with the low intersection coherence described above – makes an implementation quite problematic for certain settings.

Therefore, the intersection code is split into its computational core and into a result storage phase. The computational core is implemented entirely in SIMD mode using SSE intrinsics [Int02a]. Due to the high computational density of this code, combined with the high floating point efficiency of intrinsics-code, this part of the code never gets slower than the single-ray C-code, and thus can be safely used even if only a low degree of coherence is present.

The high cost of the result storage phase can be significantly reduced as well. Even if the overall SIMD efficiency is quite low, it often happens that either none, or all four of the rays had an intersection. Though checking these special cases separately is very much unlike typical SIMD coding, it significantly reduces the amount of the costly conditional moves. In combination, these two measures make the current SIMD intersection code well applicable in practice.

For the remainder of this chapter the intersection test of choice is the new proposed exact iterative algorithm. In general the computational cost of a single intersection test is three times higher compared to the simple linear intersection test. Nevertheless, the new method is more as twice as fast compared to Schwarze's analytic solver [Gla90] and thus an acceptable trade-off.

7.3 Efficient Memory Representation

So far, the structure of the implicit min/max kD-tree has been discussed only on an abstract level. In a naïve implementation, one would simply use the same node layout as in [WSBW01] (see Section 2.4.3), and simply add the two min/max values to each node. Assuming a default of 16-bit voxel quantities this naïve approach however requires twelve bytes for each node: eight bytes for specifying the plane and pointers, plus four bytes for the min/max values. As a kD-tree of N leaves has an additional $N - 1$ inner nodes, for N 16-bit data points $(2N - 1) \times 12$ bytes are required for the kD-tree. At two bytes per input data value, the size of the acceleration structure would then be twelve times the size of the input data.

Furthermore, the node layout assumes that the entire data set can be addressed by 29 bits [3](see [Wal04]). Larger data sets (i.e. 1024^3), would need 64-bit-pointers, or additional four bytes per node. Obviously, this 12-fold memory overhead is too high except for small data sets. For 8-bit values, the relative overhead would be even worse (20 bytes per 1 byte input data). This of course is not practical. For a practical realization it is therefore important

[3]A single 32-bit variable is used to store a nodes leaf flag (1 bit), the dimension of the splitting plane (2 bits) and a child pointer (29 bits).

to find a more memory-efficient realization.

7.3.1 Reducing Node Storage

Fortunately the memory overhead can be significantly reduced: If it is assumed for a moment that the number of voxels in each dimension is a power of two (later this constraint will be relaxed), the resulting kD-tree would be a balanced binary tree, i.e. all its leaves are on the same level. In a balanced binary tree however it is easy to show that **all** the nodes in the same level l will use the same splitting dimension d_l. Therefore, it is not necessary to store that value in each node, but just once as a single dimension-value per level.

Similarly, the split plane position has not to be stored in each node either: If level l splits $R_{x,l} \times R_{y,l} \times R_{z,l}$ voxels in the $d_l = x$ dimension, then there are only $R_{x,l} - 1$ possible split locations, and each node (i, j, k, l) [4] will use the split plane $x = x_{i,l}$. Thus, instead of storing a split in each node, only $R_{x,l}$ floats per level l will be stored in a small table. The same argument holds for $d_l = y$ and $d_l = z$ of course as well.

```
struct Level {
   //resolution of current level, i.e. 1,1,1 for root voxel
    Vec3i resolution;
    // base addr of first node in level
   Node *base;
   // split values *in* current nodes,
   //e.g.\ even the (1,1,1)-res root voxel have split = {0.5}
    float *split;
   //dimension in which current level is being split
   int dimension;
 };
```

Listing 7.2: Data structure for information about a single compressed kD-tree level.

Finally, having a balanced tree allows for performing all address computations without pointers: The address of node (i, j, k, l) is $base_l + (x + R_{x,l}(y + R_{y,l}))$, and the children of (i, j, k, l) (for splits in $d = x$ dimension) will be $(2i, j, k, l + 1)$ and $(2i + 1, j, k, l + 1)$, respectively. As a side effect of not storing any pointers, this approach will work unmodified (and even without any additional memory) also on a 64-bit architecture, and can thus handle extremely large data sets.

In summary, it is possible to get rid of all node description data except for the min/max values, thus save two thirds of the kD-tree data, and reduce the

[4] (i, j, k, l) denotes the node (i, j, k) in level l.

memory overhead from 12 to 4 (respectively from 16 to 4 for 8-bit densities).

7.3.2 Getting Rid of Leaves

Additionally to these savings, storing the min/max values for leaf nodes can be avoided as well, and instead the leaf's min/max values are computed on the fly from the cell's corner densities. This on the fly computation of the leaves is quite tolerable, as min/max operations can be implemented quite efficiently with both regular C/C++ code and SIMD extensions. Furthermore, these min/max operations have to be performed **only** for leaf traversals, which are much less common than inner node traversals. Finally, as the min/max values allow for efficiently culling non-boundary cells, almost all visited leaves also require a ray/voxel intersection, whose cost totally dominates the cheap min/max computations.

As in a binary tree half of all nodes are leaves, getting rid of the leaves allows for reducing the memory requirements by another factor of two, reducing the total overhead from 12 (respectively 20 for 8-bit values) to a mere 2. Of course, a memory overhead of two is still significant, in particular when compared to the 0.5% overhead achieved by Parker et al [PMS+99]. Nonetheless, a factor of two can be considered tolerable already, in particular as the hierarchical traversal touches only a fraction of the overall data.

Taking it all together, on average two additional values for each input data value have to be stored, i.e. N data values result in a total of $3N$ values. This factor of 3 is exactly what also many other volume rendering approaches on graphics hardware require (e.g. for storing three sets of axis-aligned slices), and is quite tolerable. Also note that only host memory is used, which is much less scarce than graphics card memory.

7.3.3 Relaxing the *power-of-two* Constraint

As mentioned before, the memory reduction scheme requires that each level of the tree has 2^i cells, i.e. that the original data set has a resolution of 2^i+1 voxel in each dimension. One simple method of making arbitrary data sets comply to this constraint would be to *pad* them to a suitable resolution.

Instead, a better solution is to *imagine* that all nodes were embedded in a larger, *virtual grid* of a suitable size that exceeds the scene's original bounding box of $[0..1]^3$, and to build the kD-tree over that virtual grid (see Figure 7.3). By properly assigning the split-plane positions, it can be guaranteed that all virtual nodes lie outside the *real* scene's bounding box of $[0..1]^3$. As the kD-tree traversal code always first clips the ray to that bounding box (see [Wal04]), rays will never be traversed outside that box, and thus can *guarantee* that no ray will ever touch any of these virtual nodes. As such, they do not have to be stored, either. Obviously, the same argument also

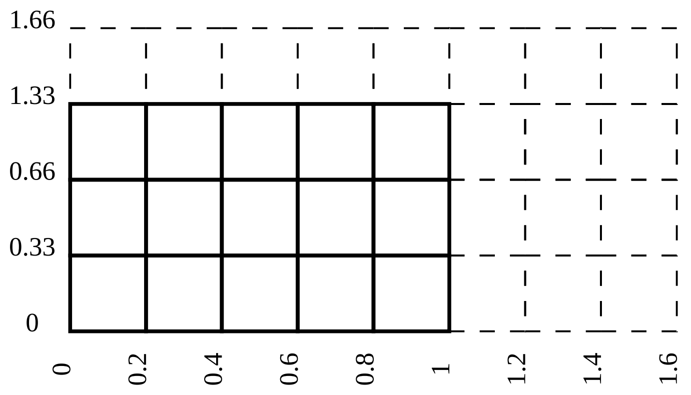

Figure 7.3: Using *virtual* nodes to relax the power-of-two constraint: This example shows a 3x5 data set embedded in a virtual 4x8 grid with a balanced kD-tree. By cleverly choosing the split plane positions it can be guaranteed that virtual voxels lie outside the scene bounds $[0..1]^2$, and thus will never be traversed by a ray. Thus, these nodes do not have to be stored, and thus do not consume any memory, either.

			Fat	Slim			
	Data	Raw		with Leaves		without Leaves	
Data Set	Bits	Data	Mem	Mem	Ratio	Mem	Ratio
Bonsai	8	16MB	316MB	64MB	5	32MB	10
Aneurism	8	16MB	316MB	64MB	5	32MB	10
ML 32^3	16	65KB	680KB	220KB	3	110KB	6
ML 128^3	16	4MB	46MB	15MB	3	7.8MB	6
ML 512^3	16	256MB	3GB	1GB	3	509MB	6
Female	12(16)	900MB	–	3.4GB	–	1.7GB	–
LLNL	8	8GB	–	36GB	–	18GB	–

Table 7.2: Memory savings of the compressed ("slim") vs. the naïve ("fat") implementation. The slim representation can achieve memory reductions of up to a factor of 10. Note that both the female and LLNL data sets cannot be rendered at all with the naïve representation, as the address bits in the fat node layout do not suffice for addressing as large data sets. The slim variant does not use any pointers at all, and thus can be used for arbitrarily sized data sets.

holds for nodes on inner levels as well.

All that has to be done to use this scheme for a data set of $R_x \times R_y \times R_z$ cells is to find $R'_{x,y,z} = min\{2^i | R_{x,y,z} \leq 2^i\}$, and just build the kD-tree over this virtually padded volume $R'_x \times R'_y \times R'_z$, while still doing the

	C			SIMD		
Data Set	Fat	Slim	Overhead	Fat	Slim	Overhead
Aneurism	1.57	0.99	1.59	3.44	2.24	1.54
Bonsai	1.79	1.14	1.57	2.91	2.1	1.39
ML 32^3	2.47	1.47	1.68	4.92	3.41	1.44
ML 128^3	1.86	1.14	1.63	2.93	2.14	1.37
ML 512^3	1.30	0.91	1.43	1.62	1.24	1.31

Table 7.3: Performance (in fps) of the compressed (”slim”) vs. the naïve (”fat”) kD-tree, for both single rays and SIMD code, measured at 512×512 pixels. Larger scenes (such as female and LLNL) could not be rendered with the fat kD-tree, due to too high memory requirements.

address computations and memory allocation with the (unpadded) original resolutions of R_x, R_y, and R_z.

Using the compression scheme, the memory overhead of the kD-tree can be reduced from 12–20 to a mere 2 (see Table 7.2), independent of the data set's resolution.

7.3.4 Traversal Overhead of the Compressed kD-tree

Unfortunately, such significant memory savings rarely come for free: Whereas the uncompressed ”fat” variant can use almost exactly the same traversal code as the original implementation [WSBW01], the compressed ”slim” variant requires additional operations in each traversal step for the address computations. In particular, it requires tracking and updating the four (i, j, k, l) indices of the current node, as well as several integer multiplications and additions for computing the children's address. Additionally, tracking a node by four indices instead of only one address requires additional stack operations. As these additional operations have to be performed for each traversal step, they can have a notable impact on total rendering performance.

As can be seen in Table 7.3, the slim variant shows an overhead of roughly 40 to 60 percent as compared to the fat variant. As expected, the overhead is slightly less for the SIMD code, as the latter allows for amortizing address computation overhead over all rays in the packet.

Overall, an overhead of at most 68 percent is quite a reasonable price for a memory reduction by a factor of up to 10. In particular for large models such as the visible female or the LLNL data set, the slim representation is the only reasonable alternative, as the high memory requirements of these scenes did not allow for rendering using the fat node layout at all. Therefore, usually the slim variant should be used, except for very small data sets.

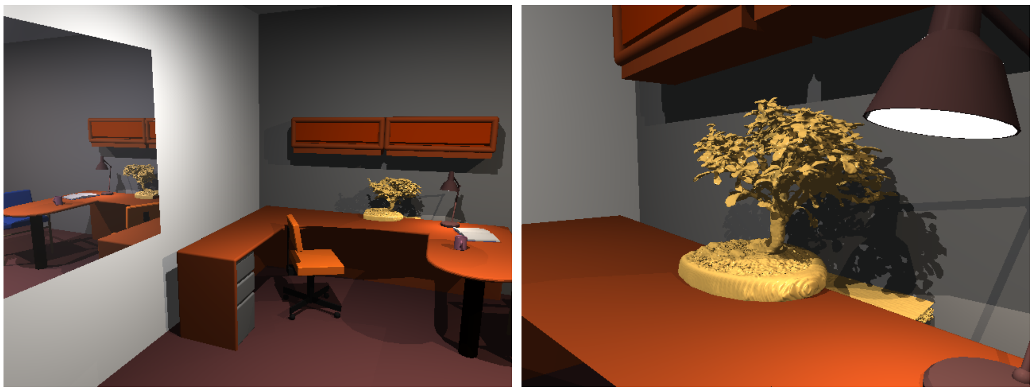

Figure 7.4: Ray tracing in a hybrid polygonal/isosurface scene, showing the bonsai data set in the polygonal office scene. Note how shadows and reflections are computed correctly between both isosurfaces and polygons. a.) Overview. b.) Zoom onto the bonsai tree. On a single PC, these scenes render at 0.9 and 1.4 fps including shadows and reflections at 640×480 pixel.

7.4 Integration into the OpenRT Engine

Using similar algorithms and data structures as the original coherent ray tracing system, the implicit kD-tree can be seamlessly integrated into the RTRT/OpenRT framework [Wal04]: In order to support dynamically changing scenes, the OpenRT system uses a two-level hierarchy in which the lower levels of the hierarchy represent polygonal meshes, which have then been efficiently organized in an upper-level kD-tree [Wal04]. This two-level structure has been modified to also support isosurface objects in the lower hierarchy level. Most core data structures (e.g. ray, hit info, shader and scene access) are shared between the polygonal and the isosurface part. Similarly, both parts share exactly the same external interface, e.g. for shooting secondary rays. Thus, it was possible to implement isosurface rendering as simply another level object.

This allowed the integration to be minimally intrusive. Few changes had to be done, and most parts of the overall system (e.g. shaders and application frontend) do not know about different object types at all. Obviously, a tight integration implies that all aspects of the OpenRT framework continue to function as before: Picking, parallelization, indirect effects like shadows and reflections, occlusion culling and early ray termination, all kinds of shaders (including even global illumination) etc. all continue to function on isosurfaces as they did on polygons. In particular, isosurfaces and polygons fit seamlessly together, i.e., a polygon may be reflected off of an isosurface,

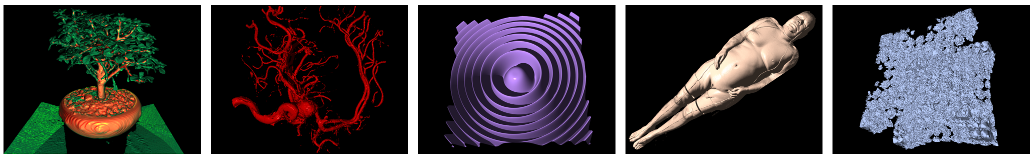

Figure 7.5: The test data sets: The bonsai tree (256^3), the Aneurism (256^3), various resolutions of the synthetic Marschner-Lobb data set (from 32^3 to 1024^3), the Visible Female ($512^2 \times 1734$), and the Lawrence-Livermore (LLNL) Richtmyer-Meshkov simulation ($2048^2 \times 1920$). These data sets have been carefully selected to cover a wide range of different data, from low (ML) to high surface frequency (bonsai, LLNL), from medical (aneurism and female) to scientific data (LLNL), and from very small (ML32) to extremely large data sets (female, LLNL).

and an isosurface may cast a shadow on any other kind of geometry (see Figure 7.4).

7.5 Experiments and Results

Once all the ingredients of the realtime isosurfacing system are described, its performance can be evaluated. Figure 7.5 shows and describes the used test data sets. In particular, it is interesting to measure its absolute performance and scalability behavior. If not mentioned otherwise, for the following experiments a *single* dual-1.8 GHz AMD Opteron 246 desktop PC with 6 GB RAM is used. Renderings are measured at a default resolution of 512×512 pixels.

7.5.1 Overall Performance Data

First of all, the overall performance of the system for different data sets is quantified. As can be seen from Table 7.4, interactive performance for all tested scenes can be achieve even on a single PC. Note that this PC is not even state of the art any more. Additionally, higher performance can be achieved by running the framework on multiple PCs in parallel. To this end a mini-cluster of five dual-1.8 GHz Opteron PCs with 2GB RAM each, linked via Gigabit Ethernet is used. Unfortunately, scalability could not be measured beyond that number, as only 5 such dual-Opterons have been available for testing. As can be seen from Table 7.4, this setup allows for frame rates of up to 39 frames per second, even including the most complex data sets. Note that this compares quite favorably to previous approaches (e.g. [NMHW02, PPL+99, DPH+03, DGP04]).

		Single PC			5-Node Cluster		
Data Set	Layout	C	SIMD	Ratio	C	SIMD	Ratio
Bonsai	fat	3.4	5.2	1.5	16.2	24.6	1.5
Aneurism	fat	3.0	6.2	2.0	14.6	29.8	2.0
ML 64^3	fat	4.3	7.8	1.8	20.1	35.7	1.7
ML 512^3	slim	1.2	2.3	1.8	6.1	11.3	1.8
Female	slim	2.7	4.2	1.5	13.6	20.7	1.5
" (zoom)	slim	2.3	7.9	3.5	11.2	39.1	3.5
LLNL	slim	0.9	1.3	1.5	–	–	–
" (zoom)	slim	1.6	5.4	3.9	7.6	28.7	3.8

Table 7.4: Overall rendering performance data when running the framework in various scenes including diffuse shading, for both a single (dual-CPU) PC, as well as with a 5-node dual-Opteron cluster. The overview of the LLNL data set could not be rendered, because the memory footprint at this view was larger than the 2GB RAM per client in the cluster setup.

7.5.2 Scalability in Data Set Complexity

In polygonal ray tracing, one of the biggest advantages of ray tracing is its sublinear (i.e., logarithmic) scalability in model size, which is due to the use of hierarchical data structures such as kD-trees [Hav01b, PMS+99].

As such a hierarchical data structure is now used as well for volumes, the same properties should also apply to the isosurface ray tracing framework. To verify this, various resolutions of the synthetic Marschner-Lobb data set are generated, and measured both the number of traversal steps and overall rendering performance. As expected, Figure 7.6 shows that the implicit min/max kD-tree exhibits roughly logarithmic scalability in model size – the slight rise of the curve beyond 256^3 is most likely due to caching effects for such large models, as can be seen by the number of traversal steps (also given in Figure 7.6) which exhibits a perfectly logarithmic behavior. This logarithmic scalability makes the method highly suitable for extremely complex datasets: Even for an increase in data set complexity from 32^3 (3.2×10^4 cells) to a full 1024^3 (10^9 cells) – corresponding to $4\frac{1}{2}$ orders of magnitude in scene complexity – the performance only drops by a mere factor of 2.1.

7.5.3 Comparison to Graphics Hardware

In order to fully appreciate this level of performance for the complex data sets, one must compare to the standard approach of extracting a polygonal isosurface to be rendered via graphics hardware. For example, a GeForce G80 currently delivers a theoretical peak performance of more than 300 mil-

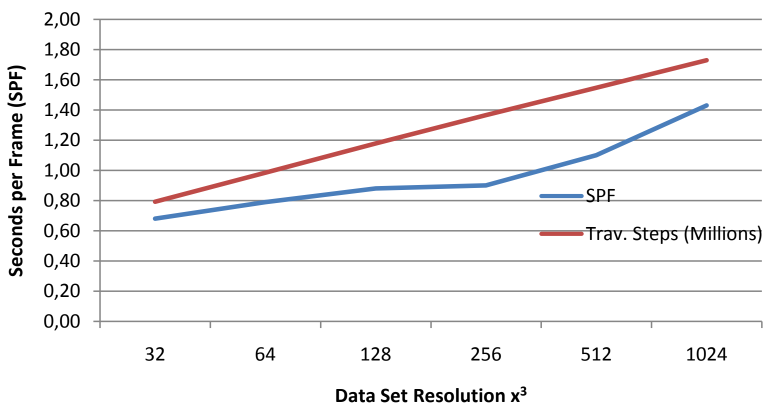

Figure 7.6: Scalability of the implicit min/max kD-tree with increasing data set resolution, measured with various resolutions of the synthetic Marschner-Lobb data set. Note the exponential scale (2^{3x}) on the x-axis, which for a roughly linear graph implies a logarithmic curve (the slight rise of the curve beyond 256^3 is most likely due to caching effects, as can be seen by the almost perfectly logarithmic number of traversal steps). Due to this logarithmic scalability, performance drops by a mere factor of 2.1 for an increase in data of $4\frac{1}{2}$ orders of magnitude.

lion shaded and lit triangles per second. However, for the LLNL data set the tessellated isosurface consists of 470 million triangles, which would require several seconds to rasterize even under best-case assumptions. Additionally, by directly ray tracing the isosurface the isovalue can still be interactively adjust, which is not easily possible using a pretessellated model. Also note that this level of performance clearly outperforms previously published isosurface ray tracing results on similar hardware.

7.6 Dynamic Updates

For some applications it is interesting to visualize time-varying data sets. Although the discussed kD-tree data structure is not specifically designed for supporting dynamic updates, in particular the slim variant is highly suitable. Assuming that the volume resolution $R_x \times R_y \times R_z$ stays constant, the slim kD-tree variant allows to support two different kinds of dynamic updates.

First of all, since the structure of the kD-tree is independent of the actual splitting plane positions the spacing of voxels in a rectilinear grid can be changed arbitrary. Only the few values in the tables that store the splitting plane positions per kD-tree level have to be updated. And second, after a modification of the actual voxel data in the volume a simple update procedure can update the already existing min/max values in the kD-tree. The last level of the tree has to fetch the real voxel data to update its min/max values, but all other nodes in the levels above can simply by updated by simple min/max operations with the values of its descendants.

7.7 Out-Of-Core Rendering

Although the current slim variant has already a small memory footprint, for massive data sets in the gigabyte range the in-core requirements might still easily exceed the available main-memory (see e.g. the LLNL data set in Table 7.2). Even if enough memory is available, if an already precomputed kD-tree has to be streamed over a slow network connection, e.g. via NFS, it can take a long time until all data is loaded and the visualization can start. The main goal of this out-of-core extension is to immediately start the exploration of the dataset without the need to wait for all, maybe multiple GB of data. Additionally, the in-core memory-footprint should be as small as possible such that even the RAM of a commodity PC is (almost) sufficient.

7.7.1 Algorithm Overview

To do so, first an LOD hierarchy of the volume is built. These LOD data are solely used for rendering as long as the data of the finest LOD level are not loaded or no more main-memory is available. Then, for each LOD level an implicit kD-tree is constructed and merged together such that a single kD-tree is obtained that is valid for all LOD levels. This merged kD-tree and the LOD data are then decomposed into *treelets* and stored in a *page-based* data structure (Section 7.8.1). During rendering, a separate thread loads all relevant treelets that are required to render the desired isosurfaces from the hard disc in a breadth-first-search (bfs) order (Section 7.8.2). Whenever a new LOD level is completely loaded, the render threads are notified such that they can use a finer LOD level for the next frame.

7.8 Out-of-Core Isosurface Rendering

After having outlined the basic approach the next sections will discuss all details about the out-of-core data structure, traversal and data loading.

7.8.1 Building the Out-of-Core Data Structure

The complete preprocessing chain is sketched in Figure 7.7. In the first step an LOD hierarchy of the volume is built. Since it is not known in advance which isovalues are interesting for the user, a simple 3D Gauss-filter kernel is used to scale down the data set, and thus it is not tried to preserve the topology of scaled down isosurfaces.

Then for each LOD level (including the original volume data) a min/max kD-tree is built. These kD-trees are merged such that a single min/max

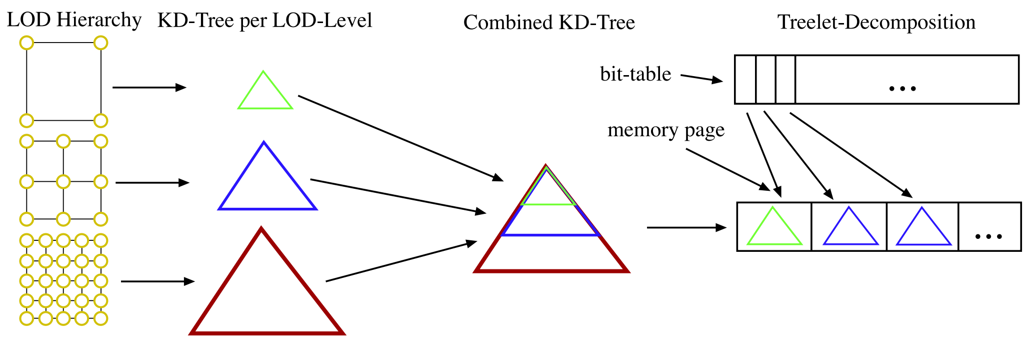

Figure 7.7: The preprocess pipeline: At first an LOD hierarchy from the volume data is constructed. Then, for each LOD data set a min/max kD-tree is built. Afterwards all LOD kD-trees are joined into a single one. This kD-tree is finally decomposed into subtrees of a certain hight and together with the corresponding LOD data of the subtree stored in a treelet array on the hard-disc. While rendering, a bit-table keeps track of what treelets are already fetched into main memory.

kD-tree is obtained. This single kD-tree could then be used to render all different LOD levels. To do so, just the min/max intervals of the nodes have to be adjusted by joining the corresponding min/max intervals. This merging is necessary because the LOD filter shifts the range of values and thus the kD-tree of the original data may not be valid for all LOD volumes.

Once the min/max kD-trees are merged, this tree and the LOD volume data are decomposed into treelets (see Figure 7.8). A treelet consists of a subtree of the min/max kD-tree with a fixed height N (e.g. 63 nodes for $N = 6$), a corresponding block of LOD voxels (similar to [BPTZ99]), or voxels from the original data set at the last treelet level, an ID that identifies the treelet, and the number of voxels in each dimension (see Figure 7.8). All subtrees have the same height except maybe the top-most subtree. Note that the number of LOD levels correlates with the height of the subtrees. If there are e.g. four levels of treelets, then also four LOD levels are used.

These treelets are stored in a page-based data structure that allows fast loading from the hard disc. Page-based means that treelets will be stored with the size of a *memory-page* on the hard disc. The size of the page is given by the operating system and is here 4k bytes. This page based treelet approach is similar to the blocklets approach by Bajaj et al. [BPTZ99] and Zhang et al. [ZN03]. If the treelets do not have the size of a page, the data will be padded to page size, or the next multiple of page size if the treelet is larger than one page. If the size of a treelet is small enough multiple treelets can be placed in one page. All treelets of a particular LOD level are stored

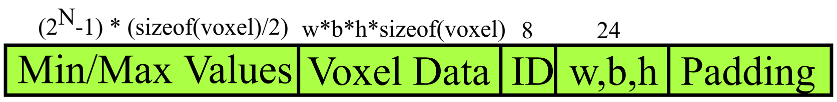

Figure 7.8: The structure and memory requirements of a treelet. Min/max values, voxel data, an ID, and the dimensions of the voxel data are grouped together. If the size of a treelet is small enough multiple treelet structures can be placed in one page on the hard disc. Width w, breadth b, and height h of the voxel block include an outer layer of voxels for calculating central differences for shading.

consecutively in breadth-first-search order.

In order to allow a proper shading, not only the voxel data of the corresponding voxel region are placed in a treelet but also an outer layer of voxels such that the central difference can be calculated for gradient estimation. In order to decrease the memory requirements, the min/max values are quantized to half of the original bit resolution. The additional traversal operations that are caused by a reduced efficiency of the subtree culling decreases the overall rendering performance typically in the range of five to ten percent.

7.8.2 Treelet Loading

For loading the required data from hard disc a similar memory-management-unit (MMU) technique as Dietrich and Wald et al. propose in [DWS05, WDS04] and [WFM$^+$05] is exploited. The MMU has to perform two tasks: The first task is to load all treelets that are required for rendering a particular isosurface without *stalling* the application and second to notify the render threads whenever all required treelets of the next LOD level are loaded.

To do so, the MMU creates a loader thread that manually fetches the data required for the current isovalue such that the loading process does not stall the render threads. The loader thread sweeps over the treelets, which are stored in breadth-first-search manner, and checks the min/max intervals of the leaf nodes of the kD-tree subtrees. If the isovalue is within the min/max interval both children will be loaded and afterwards marked as present in a bit-table. For every memory-page this bit-table has an entry that is covered by the `mmaped` (memory mapped) file with treelets. Before the sweep process starts the very first treelet is loaded separately to assure that it is available. The bit-table is necessary because it has to be known in LOD level +1 which treelets have been loaded in the previous level and thus do not touch unneeded data on the hard disc. When the isovalue is changed while inspecting the data set, it is checked if the new one lies within the current (quantized) interval. If that is not the case, the bit list is cleared and rendering starts again with the first LOD level.

In order to notify the render threads that a new LOD level is loaded, a mutex protected global variable is used. After loading a new LOD level the variable is increased such that it represents the current loaded LOD levels. The code that controls the render threads checks then before a new frame is rendered the available LODs and passes that value to its renderer threads.

7.8.3 Traversal

The basic traversal scheme for a single treelet is equivalent to the traversal of the slim-variant kD-tree. Nevertheless the overall traversal scheme has to incorporate the structure of the treelets. After having traversed a treelet it must be checked if a finer LOD level is already loaded. If yes, the address of the next treelet is computed and traversal starts again for the new treelet. If no further LOD level is available an intersection test is performed with the voxel data from the current leaf node. Figure 7.3 sketches the overall traversal algorithm.

```
treelet = LoadFirstTreelet();
StartLoaderThread();
while{1}
{
    TraverseTreelet(treelet);
    if{currentTreeletLevel < loadedLevels}
        {
            treelet = CalculateNextTreeletAddr(treelet);
            continue;
        }
        if{IntersectTreelet(treelet) == true}
        {
            return true;
        }
}
```

Listing 7.3: Traversal pseudo-code: The first treelet is loaded from the hard disc and a loader thread is started before rendering. Afterwards, for each ray the traversal starts with the first treelet and traverses as much LOD levels as have been already loaded. The loader thread updates the maximum traversal depth for the render threads whenever the necessary treelets of a new LOD level are loaded. After traversal the usual ray isosurface intersection and shading takes place with the current treelet data.

Due to the fact that all needed data for treelet traversal, intersection tests etc. are stored in one memory page cache-aliasing can be reduced (since data in a single page map to different sets in the cache, see also Section 5.4.1).

Another advantage is that the height of the subtree into a treelet is known in advance. This fact can be used to remove a conditional (leaf-node check) in the innermost traversal loop.

7.9 Results

In order to evaluate the efficiency of the out-of-core data structure performance numbers for two data sets are measured: Time-step 270 of the LLNL Richtmyer-Meshkov instability with a voxel resolution of $2048^2 \times 1920$ as well as a synthetic data set Attractor (see Figure 7.9).

The Attractor volume consists of three different 3D-attractors and a voxel resolution of 2048^3. Both data sets use 8-bit data values. The test system is a dual dual-core Opteron 275 (2.2GHz) PC with 8GB main memory. Image resolution is always 640×480 pixel and for shading a simple diffuse surface-shader is used (see Table 7.6).

7.9.1 Treelet-Size Influence

As described in Section 7.8.1 the LOD data and min/max kD-tree is decomposed in a treelet hierarchy. The number of treelet levels influences the required disc space, in-core RAM for a particular isosurface, loading time, and rendering performance. Table 7.5 shows the results when the height of the treelets is increased, and thus reduces the number of levels in the treelet hierarchy.

The preprocessing time in Table 7.5 includes loading/writing the data from/to hard disc, LOD creation, min/max kD-tree building, and merging. In the experiments a treelet height of nine resulted in the best overall performance numbers – at least for the tested data sets. With larger treelets, some numbers i.e. the required in-core RAM increase again.

Treelet Height	preprocess Time (h)	Disc Space (GB)	Working Set (GB)	Loading Time (Min)	FPS
3	17	292	16.0	58	1.0
6	3	66	10.0	30	1.8
9	1	33	6.1	5	2.5

Table 7.5: Performance numbers for the LLNL data set with increasing treelet heights. If the treelet height is increased, the preprocessing time, the required in-core memory for rendering a particular isosurface, and the loading time of the relevant data decrease significantly. At the same time the rendering performance almost doubles.

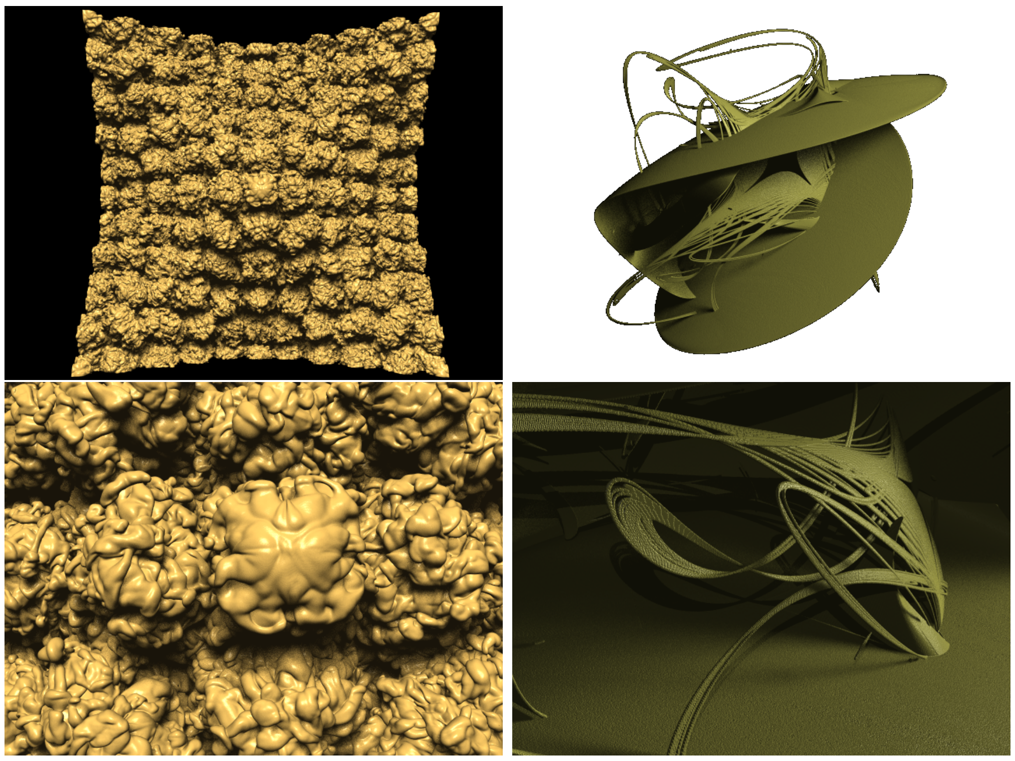

Figure 7.9: Example images of the test scenes: LLNL and Attractor rendered with phong-shading, progressive soft-shadows and a single point-light source between 1.0 and 1.9 fps at 640×480 image resolution using two CPU cores.

7.9.2 Overall Rendering Performance

The overall rendering performance is presented in Table 7.6 (see Figure 7.9). As the fps numbers show always at least interactive rendering performance is achieved. These numbers indicate the FPS for the finest level in the hierarchy. The rendering of coarser LOD levels is faster and drops down with every finer LOD level.

Furthermore, the required in-core memory is reasonable, especially if it is considered that quantized min/max values in the treelets are used. That means that not only treelets are loaded for a single isosurface but all treelets for a quantized "bucket".

7.9.3 Comparison

In comparison to the approach of DeMarle et al. [DPH+03] almost the same rendering performance is achieved by using only a single PC rather then a 32 PC cluster setup (although obviously a newer and correspondingly more powerful multi-core CPU is used compared to the CPUs used in DeMarle's

Scene	Iso	Loading Time (Min)	Working Set (GB)	FPS 2 Cores	FPS 4 Cores
LLNL (zoom)	16	5	6.1	2.0	3.9
LLNL (overview)	16	5	6.1	2.6	5.1
Attr (zoom)	25	4	2.1	1.5	2.9
Attr (overview)	25	4	2.1	3.6	6.9

Table 7.6: Overall rendering performance for the two test scenes (Figure 7.9) measured with a simple diffuse shading using two and four CPU cores. As the results show, independent from the viewpoint at least interactive frame rates can be achieved. The loading time is the time until all relevant data from the finest level are loaded.

cluster). This reduces the hardware requirements significantly.

In addition, very fast loading times can be achieved due to the linear memory-page loading from hard disc. Wald et al's [WFM$^+$05] view dependent out-of-core method requires for the LLNL data set approximately 18 minutes until all relevant data are loaded. However, only the visible data are loaded and if the camera position changes loading starts again. This is in contrast to the presented method were all relevant data for an isosurface are loaded independent of visibility. Furthermore, the in-core footprint is smaller: 6.1GB in the new system compared to 8.0GB in Wald's approach (for all views). An additional plus is the faster kD-tree traversal. Due to the simple out-of-core scheme approximately twice the rendering performance of Wald's approach can be achieved.

Compared to the octree approach of Knoll et al. [KWPH06, KHW07] the rendering performance is higher but on the other side more main-memory is required due to the usage of a kD-tree. Nevertheless, both approaches have orthogonal feature sets (except that Knolls approach does not easily support dynamic updates because the octree is stored in a compressed format). Both approaches could be combined into a single system either by adding their in-core octree compression scheme and view-depedent LOD usage into this system or by extending their approach with the presented out-of-core approach.

7.10 Applications

After having described both the framework and analyzed its performance, some of the practical applications will be discussed that it allows for.

7.10.1 Interactive Exploration of Complex Isosurfaces

Due to the logarithmic scalability in data set size (see Section 7.5.2), one obvious application of the framework is the interactive visualization of highly complex data sets. For example, Figure 7.10 shows the $512 \times 512 \times 1920$ Visible Female data set, rendered with different shader configurations. Except for the transparent skin example, fast SIMD code can be used for visualizing the model, and achieve frame rates of 8.6, 5.0, and 4.0 frames per second at 640×480 pixels, respectively, even on a single PC. Due to splitting up of the rays, for the transparent skin example single-ray code is exploited, but still – including all secondary rays – achieve 0.8 frames per second per PC. Using the distribution features of OpenRT, higher frame rates can easily be achieved by running the system in parallel (see Table 7.4).

7.10.2 Interactive Global Illumination on Isosurfaces

Once being able to handle shadows and reflections, it is an obvious next step to also support global illumination on isosurfaces. For that purpose, the "Instant Global Illumination" technique [WKB$^+$02, BWS03b, Wal04] is used, in which the illumination in a scene is approximated using "Virtual Point Lights" generated by tracing light particles into a scene and using those for illumination.

The Instant Global Illumination algorithm is completely independent of geometry, only requires the ability to shoot rays, and thus is ideally suited for the hybrid polygon/isosurface setting. As four-ray packet-traversal is supported, even the fast implementation of Benthin et al. [BWS03b] could be used without major modifications.

Figure 7.11 once again shows the bonsai model on the desk of the office scene, now with global illumination from three area lights turned on. As can be seen, all indirect interactions between the polygonal scene and the isosurface data set work as expected.

7.11 Conclusions and Future Work

In this chapter, it was shown how advancements in polygonal ray tracing can be leveraged to also significantly increase interactive isosurface ray tracing performance on off-the-shelf PCs.

To this end, the usage of an "implicit kD-tree" for storing the data set in a hierarchical way is proposed that is well suited for efficient ray traversal. Furthermore a memory efficient representation has been discussed which was also extended to support out-of-core data sets. The presented data structure and proposed novel exact isosurface intersection test together allow for achieving interactive isosurface ray tracing performance on individual PCs, and furthermore allow for scaling performance by running in parallel on mul-

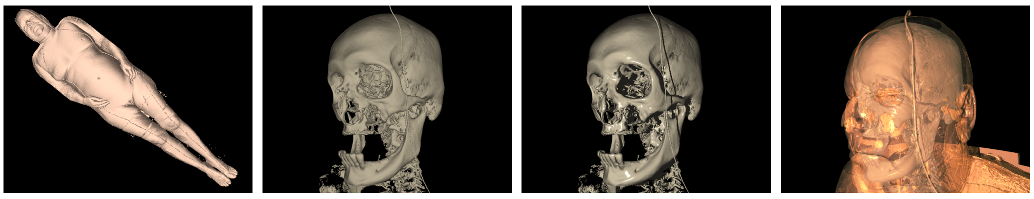

Figure 7.10: The Visible Female (512 × 512 × 1920), rendered at 640 × 480 pixels on a single dual-1.8 GHz Opteron PC a) Overall model with direct display of the skin isosurface (8.6fps/1PC). b) Zoom onto the head with bones isovalue (5FPS/1PC). c) with additional shadows (4FPS/1PC). d.) The same, plus semi-transparent skin (0.8fps/1PC).

tiple PCs. Even for highly non-trivial data sets, interactive performance can be achieved on a single dual-processor desktop PC. Due to the excelent scalability in data set complexity, this level of performance can be maintained even for massively complex data sets of several Gigabytes.

The new hierarchical kD-tree based data structure, as well as the optimized implementation allow to clearly outperform previously published isosurface ray tracing approaches in particular when considering the flexibility of the overall system. While on a single PC GPU-based methods can achieve higher frame rates for small datasets, they usually do not easily scale to larger datasets, and for datasets as used in the presented system, they are often not

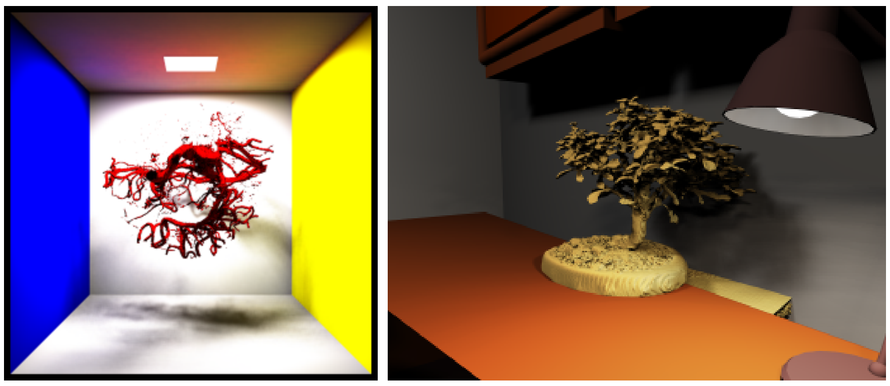

Figure 7.11: Instant global illumination on isosurfaces. a) The aneurism data set in a Cornell box. Note the slight color bleeding on the ceiling, as well as the smooth shadows on the walls and the floor. b) Bonsai tree in the office scene, with smooth shadows and indirect illumination. As the method is tightly integrated into the RTRT/OpenRT system, the Instant Global Illumination implementation can be applied to the isosurfaces just as easily as originally proposed for polygons.

applicable at all. Note however that the proposed methods are not limited to CPUs, but should similarly benefit GPU-based ray tracing approaches (e.g. [WS01, GPSS07, PGSS07]) as well.

Having never made any assumption on the isovalue, the isovalue can be interactively changed any time, and even multiple isosurfaces of the same data set can be easily rendered concurrently. Finally, being tightly integrated into the OpenRT engine, the presented framework allows for augmenting isosurfaces with ray traced lighting effects such as transparency, shadows, reflections, refraction, and even global illumination. At the given level of performance, all these effects can be fully recomputed every frame even under interactive changes to camera, isovalue(s), or scene.

In a next step, it would be interesting to investigate how to further reduce the memory overhead, for both main memory and hard disc, in order to have a comparable memory-footprint e.g. to [KWPH06]. While the current implementation is already quite fast, there is still room for improvement. In particular, the exact caching behavior requires closer attention, in particular for complex data sets.

Finally, it is an obvious next challenge to investigate complex time-varying data sets such as the full 1.5TB LLNL dataset. In particular the hierarchical nature of the proposed approach seems promising for this specific application.

Chapter 8

Isosurface Ray Tracing of Tetrahedral Volumes

In this chapter, a new approach is proposed to directly ray trace isosurfaces defined over tetrahedral domains by combining recent advancements in polygonal ray tracing with existing techniques for isosurface extraction. A novel ray-packet tetrahedron intersection test, inspired by the marching tetrahedra algorithm, and its integration with a coherent implicit bounding volume hierarchy traversal is detailed. These techniques are extended to time-varying data sets as well as practical shading and visualization features such as multiple transparent isosurfaces and dynamic shadows.

8.1 Isosurface Ray Tracing of Tetrahedral Meshes

The core of this novel approach to ray trace unstructured scalar fields, that are decomposed into tetrahedral meshes, is an implicit dynamic bounding volume hierarchy in the spirit of implicit kD-trees (see Chapter 7). This is used with an aggressive coherent ray traversal, and a specially designed ray-packet isosurface intersection technique inspired by fast packet-triangle intersectors and the Marching Tetrahedra [DK91a] algorithm.

In unstructured volumetric grids, the scalar field can be defined through linear interpolation over tetrahedral primitives; each such tetrahedron can then contain one or more more isosurfaces given user-specified isovalues. As with implicit kD-trees, a hierarchical index structure is built over these primitives such that each node in the hierarchy contains the minimum and maximum of the scalar field below that node's subtree. These min/max intervals, or isoranges, can then be used during traversal to discard subtrees that cannot contain the isovalue. Nevertheless, instead of kD-trees, bounding volume hierarchies are now used. In practice, they are at least as fast, equally efficient for time-varying data, and better suited to the irregular, overlapping geometry of unstructured volumes.

The implicit bounding volume hierarchy encourages a variation of the aggressive packet-frustum BVH traversal that was recently proposed for polygonal ray tracing [WBS07]. This traversal operates on much larger packets (typically 8x8 or 16x16 rays) than the 4-ray SIMD traversal used for implicit kD-trees, and uses frustum culling and speculative descent to minimize the number of ray-node traversal steps. Larger packets also imply better amortization of per-packet costs, and thus help in hiding the overhead induced through implicit culling. Since the implicit BVH is built over the space of all isovalues, the isovalue(s) of interest can be changed interactively any time, and even multiple isovalues can be trivially supported. A BVH also allows for easily updating the data structure once the scalar field or even vertex positions change, and thus allows for naturally supporting time-varying data.

In both core algorithms intersection and traversal, heavy use of large-packet/frustum techniques is made which were recently developed in polygonal ray tracing. This large ray packets can be supported here since it is not aimed to integrate this new techniques into the OpenRT system. Unless otherwise specified, both intersection and traversal are assumed to operate on packets of 16×16 rays.

8.2 Isosurface Intersection

An isosurface is the implicit surface $f(x, y, z) = c$ where a scalar field $f(x, y, z)$ takes on a given isovalue c. For conventional first-order finite elements, the scalar field is given as a tetrahedral mesh in which the scalar values specified at the vertices A, B, C, and D; the scalar field inside each *isotetrahedron*, or *isotet*, is defined by linear interpolation $f(x, y, z) = \alpha A + \beta B + \gamma C + \delta D$, where $\alpha, \beta, \gamma, \delta$ are the barycentric coordinates of (x, y, z).

To intersect a ray $\mathcal{R}(t) = \mathcal{O} + t\vec{\mathcal{D}}$ with any isosurface $f(x, y, z) = c$ one can immediately substitute the ray equation into the linear interpolation equation, solve a resulting linear system for t, and check that the solution lies within the isotet[1]. However, it can be observed that for linear interpolation an isosurface must be planar within a tet. This plane is bounded by line segments along the edges of the isotet in which it exists, forming either a triangular or quadrilateral polygon as shown in the various cases of Marching Tetrahedra. This extracted polygon is denoted an *isopolygon* (or *isopoly*) for the rest of this chapter. Unlike solving the ray-parametrized implicit, this isopolygon must only be computed once per isotet traversed; that cost is amortized over all rays in the packet, and the full array of fast ray-polygon techniques can be applied.

[1]Please note that A, B, C, D are 4D coordinates denoting the 3D space position plus the voxel value at this position as 4th component. Similarly the ray has to be extend to 4D with $\mathcal{O} = (x_o, y_o, z_o, c)$ and $\mathcal{D} = (x_d, y_d, z_d, 0)$.

8.2.1 Extracting the Isopolygon

To compute the plane equation and bounding edges of the isopolygon, the Marching Tetrahedra algorithm [DK91b] is used. Vertices of the isopolygon lie on edges of the isotet, and isopolygon edges lie on the tet faces. Polygon vertices will lie only on those tet edges for which one vertex is greater and one is smaller than the isovalue. Having four vertices, there are only 16 cases for which a given vertex is either larger or smaller than the isovalue. For each of these cases, it can be stored how many vertices the resulting polygon will have, and the indices of the two tet vertices that span the edge on which that polygon vertex must lie. In SSE, this lookup is particularly simple: after loading the four vertices' isovalues v_i into a SIMD register, a single SSE comparison, $\forall v_i > c$, followed by a `movemask` operation will return conveniently the case in a 4-bit integer (one bit for each comparison) that can be directly used to index into aforementioned table of 16 cases. Once it is known which tet edges contain an isopolygon vertex, each isopoly vertex can be computed by linear interpolation along the two vertices of the corresponding tet edge i.e.:

$$V = P\frac{c - P_v}{P_v + Q_v} + Q(1 - \frac{c - P_v}{P_v + Q_v}), \tag{8.1}$$

where P and Q are two (4D) tet vertices, and V is the resulting isopolygon vertex.

8.2.2 Ray-Isopolygon Intersection

Once the vertices of the polygon are known, an extension of Wald's triangle test [Wal04] can be used to intersect it. As shown in Figure 8.1 (left), ray-isopolygon intersection first computes the distance to the precomputed plane, then projects the ray hit point onto a suitable 2D coordinate plane. Here, each of the edges defines a (2D) half-space, which is oriented to point towards the inside of the isopolygon. Since the isopolygon must be convex, the projected hit point can then be taken and perform a 2D half-space test with each of the edges, and can reject the hit point as soon as any of these tests fails. This test can be performed efficiently for four rays in SSE for both triangle and quad cases.

8.2.3 SIMD Frustum Culling

In addition to fast SIMD intersection, a conservative "full miss" and "full hit" tests is also applied for the entire packet, using packet frustum culling, e.g. [DHS04, BWS06b]. These tests require computation of the four corner rays bounding the packet frustum in SSE. For a given isopolygon, individual ray intersections can be avoided when all four bounding rays fail for the *same*

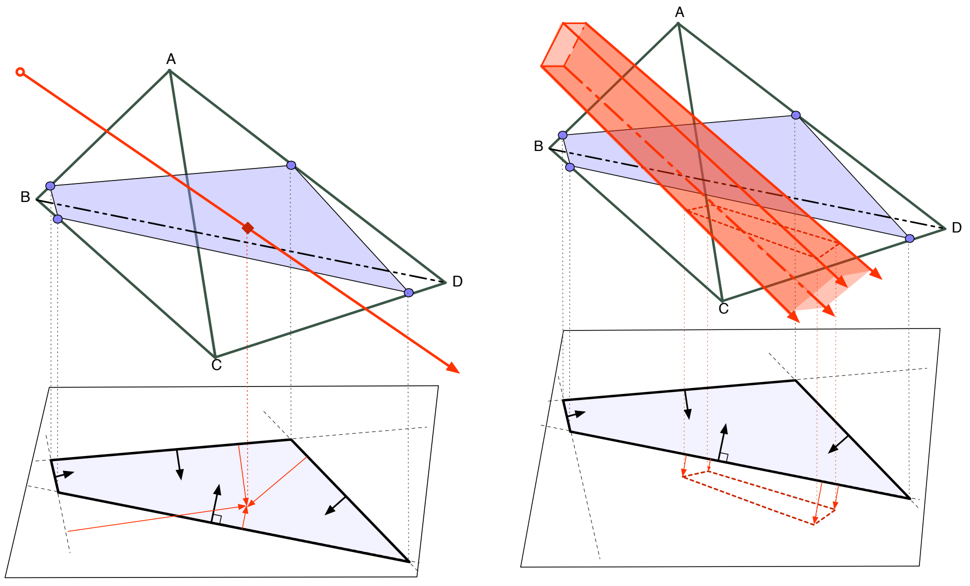

Figure 8.1: Ray-Isopolygon Intersection in an Isotetrahedron: Knowing that the isosurface inside the tetrahedron is a plane, first an isopolygon is extracted. Then the point where the ray pierces that polygon's supporting plane is computed, and project both the polygon and that hit point to a 2D coordinate plane. In 2D, then a point in (convex) polygon test is performed by considering if the point is on each of the edges' positive half-spaces. The test can trivially be extended to support frustum culling: If all corner rays of the bounding frustum fail at the *same* edge, all the rays inside the frustum must fail.

2D half-space test (Figure 8.1, right). Similarly, if all four rays pass all half-space tests, the entire packet passes through the triangle, and it must be only a distance test performed for the component rays. Thus, intersection tests for individual rays are only required when the frustum neither fully misses nor fully hits.

The efficiency of frustum culling depends on the relative areas of the frustum and isopolygon within the plane. For complex scenes, tets are too small to have full hits, and frustum culling rarely succeeds. However, full misses are quite common due to the loose nature of the implicit BVH, making this test highly effective overall. Typically, frustum culling can reject 40–60% of the packet-isopolygon tests, tough this ratio declines for larger models. Every time SIMD frustum culling rejects a packet test, 256 individual ray-isopolygon tests are avoided.

8.2.4 Isopolygon Precomputations

Isopolygon computation can be executed in three ways:

1. **Full precomputation:** Precompute all isopolys every time the user changes the isovalue(s) of interest.
2. **On-the-fly computation:** from scratch on demand.
3. **On-the-fly computation with caching:** Compute isopolys only when needed, but keep a cache of already computed isotets; clear the cache every time the user changes the isovalue(s) or time step.

Full precomputation maximizes performance for navigation with static isovalues, but requires a larger memory footprint and incurs delays when the user changes isovalue or time step in time-varying data sets. On-the-fly computation is slower during rendering, but offers greater flexibility with scene interaction. Caching in theory offers a compromise, but in practice is quite complicated in a multi-core environment, as it requires the resolution of cache conflicts in a thread-safe manner, requiring significant synchronization overhead. Therefore only on-the-fly computation is used by default. Due to the use of large packets – which allow for amortizing the on-the-fly computations over 64 rays – the overhead is in the range of 5–8%, which is a tolerable price for the ability to arbitrarily change the time step or isovalue.

8.3 Shading Normal Interpolation

After determining the proper hit points for a ray bundle (if any exists) for each pixel a corresponding color has to be computed. For many shading models i.e. Phong [Pho75] a shading normal needs to be interpolated at the hit point.

To support smooth interpolated normals at each vertex position a gradient could be estimated in a preprocess (or on-the-fly). This gradients would then be used in the surface shaders for normal interpolation by: $n(p) = \alpha n_1 + \beta n_2 + \gamma n_3 + \delta n_4$ with $\alpha, \beta, \gamma, \delta$ being the barycentric coordinates of the hit point p.

Unfortunately, the described isopoly intersection test does not calculate the barycentric coordinates within the isotet. To circumvent this problem the gradients are not stored directly but an intermediate 4×4 matrix is determined for each isotet. This matrix can then be used to calculate an interpolated normal at p *without* the actual vertex gradients and barycentric coordinates $\alpha, \beta, \gamma, \delta$.

To do so, two matrices are set up: $V = (A, B, C, D)$ and $N = (n_1, n_2, n_3, n_4)$ with A, B, C, D as the 4D vertices of the isotet and n_1, n_2, n_3, n_4 the corresponding gradients. Now the intermediate matrix $M = NV^{-1}$ can be calculated and stored. Finally, for any given p inside the isotet Mp will yield the interpolated normal. Of course this comes at the price of storing a complete 3×4 matrix for each isotet (the last row has always the form $(0, 0, 0, 1)$). However, this allows us to interpolate a normal without the need of computing the barycentric coordinates directly.

8.4 The Implicit Bounding Volume Hierarchy

The concept of the implicit BVH is similar to that of the implicit kD-tree (see Chapter 7) in that the acceleration structure is not built for a single isovalue, but rather as a tree of min-max isovalue ranges (e.g. Wilhelms & Van Gelder [WG92]). Each node stores the minimum and maximum of all scalar field values contained within that subtree. During traversal, all BVH nodes that do not contain the desired isovalue can be consequently culled. Once built, the implicit BVH structure is valid for all isovalues, and thus allows for simultaneously rendering multiple isosurfaces from the entire range of isovalues. As subtrees that do not contain the isovalue are never traversed, the only effective cost of supporting arbitrary isovalues is a slightly looser-fitting BVH.

8.4.1 Building the BVH

Building an implicit BVH for tets in fact is similar to building a BVH for triangle meshes. Most mesh-BVH builds rely on bounding boxes or centroids of their primitives as construction metrics (Section 2.4.2), and tets behave similarly to triangles in this regard.

Traditional bottom-up BVH builds (e.g. [GS87]) generally result in inefficient BVHs [Hav01a]. Recent BVH literature has favored top-down builds, which recursively partition primitives into two subgroups. Two partitioning strategies are of particular interest: Wald et al.'s sweep surface area heuristic (SAH) build [WBS07] and Wächter et al.'s fast spatial median build as proposed in his bounding interval hierarchy paper [WK06]. The SAH build employs a *surface area heuristic* [GS87, Hav01a] to select a partition with lowest expected cost, but is costly to build. The BIH-style build is closer in spirit to spatial median builds and, as it requires no cost function evaluation, it builds significantly faster than SAH methods. In both constructions, nodes are partitioned until leaves contain 12 or fewer tet primitives. Empirically, experiments showed that this fixed value works best.

BVH Structure: The BVH node employs the same structure as [WBS07], with a crucial modification: the minimum and maximum voxel values of a subtree are interpreted as 4th dimension of the axis-aligned bounding volume defined by b_{min} and b_{max}, leading to 4D bounds $b_{min} = \{x_{min}, y_{min}, z_{min}, v_{min}\}$ and $b_{max} = \{x_{max}, y_{max}, z_{max}, v_{max}\}$. These can then be stored and processed per node as SSE vectors. Integers for the child node index and traversal bookkeeping follow, padded to ensure SSE-friendly 16-byte alignment. Storing isovalues alongside geometric extents allow all dimensions to be processed simultaneously in SSE.

8.4.2 Implicit BVH Traversal

Having described the construction of the implicit BVH, now the traversal procedure is detailed. As previously mentioned, the coherent traversal algorithm of Wald et al. [WBS07] is employed, and extended with implicit iso range culling. In general, this algorithm operates on large packets of rays, and tracks both a bounding frustum and the first "active" ray in the packet that intersects a current BVH node. Instead of intersecting each traversed node with **all** the rays in the packet, it employs optimizations such as speculative descent and frustum culling of nodes. With the implicit BVH, nodes not containing an isovalue in their min-max range are culled. More generally, these traversal tests proceed as follows:

I) Implicit Culling: At the heart of implicit BVH traversal lies the concept of culling subtrees that are known to be *inactive* – those whose isorange does not contain an isovalue. As this test is very cheap, it is performed first. In addition, it can be observed that each active node must have at least one active child, and if the first child is inactive, it can be continued with its active sibling. Only at *bifurcation nodes* - where both children are active - it is actually reverted to the geometric tests outlined below. In the worst case, this behavior descends several times into a subtree that is not actually visible. Since these speculative descents are fast, however, this is still quicker than testing all the nodes for visibility; and even *if* the fast descent led to a subtree that is outside the packet's bounding frustum, this node would be immediately rejected by the frustum test outlined below (see Figure 8.2).

II) Speculative First-Active Descent: For the first geometric traversal test, the first *active* ray is examined in the packet. If that hits the current node, it can be immediately descend without performing any more ray-box tests, as illustrated in Figure 2.7(a). Since it is never tested whether any of the other rays actually hit the current node, this test is speculative. Though it may cause modest extra work when few rays in the packet are also active, this strategy allows many ray-box tests to be skipped when numerous consecutive rays are active.

III) Frustum Test: If the first active test fails, it is sure that the packet at least partially misses the box, and a frustum test to conservatively

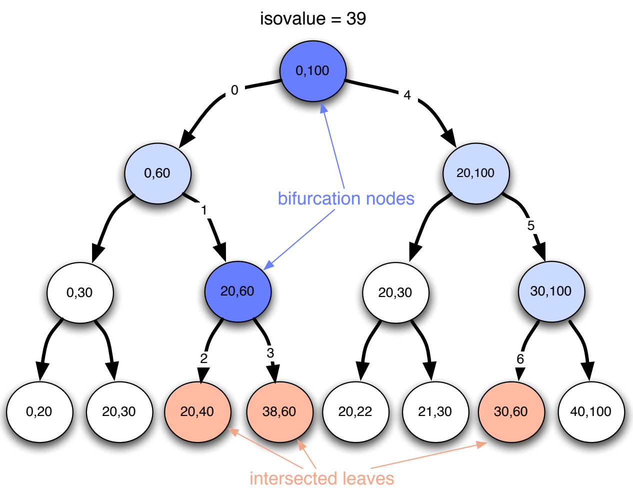

Figure 8.2: *Implicit Culling.* The implicit BVH is a min-max tree containing only a subset of BVH nodes containing the desired isovalue(s). Speculatively it can be descend the min-max tree until a leaf is reached, or an intersection test fails. Only at bifurcation nodes (dark blue) it must be switched immediately to geometric packet-BVH traversal computation. Thus, geometric tests are performed as if the BVH had only been built over active nodes for a single isovalue.

determine if the entire packet misses is performed. Technically an interval arithmetic (e.g. [RSH05, BWS06b]) test is employed instead of a geometric frustum test, but the effect is similar in behavior. If the full packet missed, the current node is rejected and the next node on the stack is processed (see Figure 2.7(b)).

IV) First-Active Ray Tracking: If both the speculative descent and frustum tests fail, all remaining rays are tested until the first active one is found that hits the current node. Those rays that failed the test are marked inactive by tracking the index of the first active ray in the packet (all rays with a smaller index are known to be inactive). If no active ray could be found, the node is rejected and the next subtree is popped from the traversal stack. Rays with indices higher than the first active one found so far are not tested, and are speculatively descended into the subtree as well.

V) Leaf Traversal: When encountering a leaf, first a frustum test is performed as for all other nodes. If that test passes, all tets referenced in that node are visited sequentially. For each tet then the isorange is determined (which may be smaller than the node's isorange), and used to test that range against the isovalue. Finally the tet is either rejected or intersected as described above.

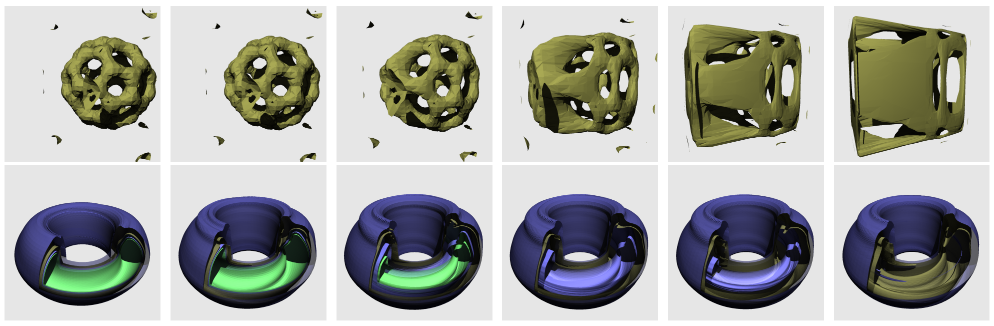

Figure 8.3: Two examples of time-varying data sets, rendered at 1024×1024 pixels, using a 16-core 2.4 GHz Opteron workstation. Top: An artificially created deforming bucky ball that shows severe deformation of its 226K tets, running at 50+ frames per second including shadows from a point light source. Bottom: The fusion data set with a time-varying scalar field (3m tets, 116 time steps), rendered with four layers of isosurfaces, a crop box, shadows, and transparency, running at 7 to 15 frames per second. Camera and light positions, time step, and number and parameters of the isosurfaces can be changed interactively.

8.5 Time-Varying Data

Time-varying data is extremely common in FE (finite element) simulations. In the simplest time-varying tet meshes, geometry remains constant and only scalar values change. More complex scenarios include changing geometry and topology, and potentially dynamic addition and removal of elements from one time step to the next. To address these possibilities, two schema for dynamic BVH construction, balancing performance and memory footprint are proposed. Results are analyzed in Section 8.7.5.

8.5.1 Schema I: Unique BVH Per-Step

The naïve way of accommodating time-varying data is to compute a unique BVH for each individual time step. Thus, no render-time computation is necessary to progress from one time step to the next, regardless of changes in geometry or scalar element values. As only main memory is used, this approach is in fact very efficient. However, for large data sets with many time steps such as the fusion data set in Figure 8.3, this approach may require considerable amounts of memory which might not be available.

8.5.2 Schema II: Dynamic Refitting

Fully computing a new BVH on-the-fly during rendering is too costly for large data, even using the fast BIH-style build. However, it can be observed that when tet mesh vertices change position but connectivity remains constant, the BVH structure will not change drastically between time steps. Thus, simply refitting the nodes' bounding extents will yield a correct BVH. This technique has been successfully applied to ray tracing dynamic triangle meshes [WBS07, LYTM06]. The main drawback is that, particularly in cases of extreme geometric deformation, the refit BVH may perform worse than a BVH built from scratch for that particular time step. Fortunately, for tet meshes and the BVH, this method works extremely well due to the continuous nature of tet deformations in FE simulation, particularly for rigid bodies. Moreover, when vertices remain constant but the scalar field changes, the BVH is identical for all time steps, as only the min-max isovalues must be updated.

As previously mentioned, minimum and maximum geometric bounds and isovalues are stored adjacently in 4D SSE vectors. Refitting the 4D extents can thus be accomplished with one SSE min and one SSE max per BVH node. Tet vertices and scalars are also stored as 4D points; thus computing the 4D bounds of a tet is also extremely efficient, requiring only 3 SSE min and max operations each per tet. With multi-core CPU's, it is straightforward to parallelize the update process. After the initial BVH has been built, for each refit thread an entry point in the BVH is computed such that the number of nodes in its subtree is approximately equal for each thread. During a refit, these subtrees can then be update in parallel. Once all subtrees are updated, a single thread refits the remaining few nodes close to the root node.

8.6 Shading and Interaction Modalities

Having leveraged the algorithms for efficient unstructured volume ray tracing, several visualization modalities will now be described that can assist in understanding unstructured data sets.

Smooth Normals: Since linear interpolation in tetrahedral meshes leads to piecewise-planar isosurfaces, the rendered isosurface has normal discontinuities where different tets abut, resulting in a faceted surface appearance. Instead of using the geometric surface normal for shading, a smooth surface appearance can be achieved by precomputing and interpolating vertex normals, with little additional cost (see Section 8.3). Though visually more pleasing, most scientists prefer seeing the data "as computed", so this feature is by default disabled.

Shadows: A far more useful effect that a ray tracer can support is shadows, which can add important visual cues over an object's shape (see Figure 8.5). In casting shadow packets, rays are generally coherent and share a common origin in the case of point lights. Unlike primary rays, shadow rays do not inherently form a regular beam, and thus have no concept of "corner rays"

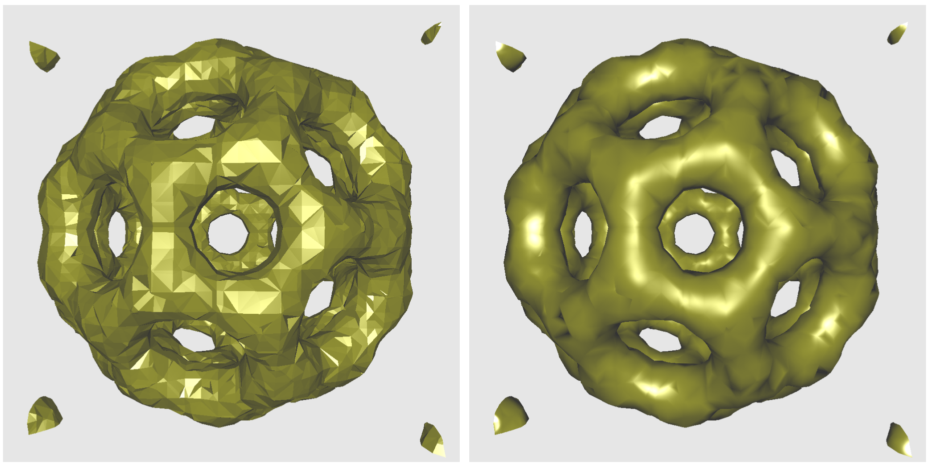

Figure 8.4: Instead of shading with the surface normal, a smoother appearance can be achieved by precomputing and interpolating the tets' vertex gradients.

for SIMD frustum culling. Though bounding corner rays could easily be computed from a packet [BWS06b], this is not yet implemented. Fortunately, frustum culling during BVH traversal still works for shadow rays using the Reshetov et al. [RSH05] technique, which requires no actual geometric frustum. In theory, shadow rays are simpler than other rays, as they can be terminated as soon as *any* valid intersection is detected. These special cases, however, are not yet exploited. The overall speed impact of shadow rays varies, but is typically lower than 2× (see Figure 8.5a-b).

Multiple Isosurfaces: Supporting multiple isosurfaces in an implicit BVH is straightforward, by simply testing whether a BVH subtree overlaps *any* of the isovalues before descending it. To follow the SIMD paradigm, up to four different isosurfaces are supported, though it would be trivial to add more. Keeping the four isovalues in a SIMD vector, it can then be tested whether a BVH node's or isotetrahedron's iso range contains any of these four isovalues in parallel. These are in turn intersected with all the rays that actually hit the leaf node. Though rendering multiple surfaces can require tracing more rays per image, particularly when transparency is enabled, it causes no significant computation penalty in and of itself.

Clipping Planes and Boxes: While isosurfaces provide an intuitive way of visualizing a data set, one of their drawbacks is that the surface often occludes the data set's interior. For that reason, visualization systems often employ clipping planes (or boxes) that allow for cropping certain parts of the model to expose its interior. Currently a single box that may or may not

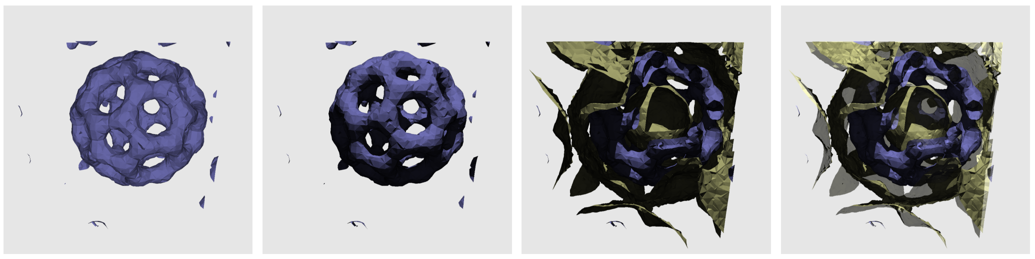

Figure 8.5: Impact of adding additional shading effects: a) A bucky ball rendered with a single isosurface, and diffuse shading. b) After turning on diffuse shading with shadows. c) With a second isosurface and an interactive clip-box to expose the interior. d) Adding transparency as well. At 1024 × 1024 pixels on a Intel Core 1 duo laptop, these screenshots render at 15.6, 10.2, 5.4, and 2.6 frames per second, respectively. On a 16-core Opteron 2.4 GHz workstation, they render at 90, 70, 42, and 19 frames per second, respectively.

extend to infinity (to simulate a plane) is allowed for, and use this to clip BVH subtrees. During traversal, if a node's subtree is completely enclosed in the crop box, the subtree is skipped just as if it was out of the isorange. In SIMD, a box-in-box test is very cheap and can be amortized per packet, incurring negligible cost. An example of this feature is shown in Figure 8.5.

Transparent Depth Peeling: Another effect that allows one to see through an isosurface is to render it with transparency. Though straightforward to implement, transparency multiplies the complexity of rendering an image by the number of transparent hits required. Depth peeling could also be handled by storing multiple hit points in a ray packet, but for the currently used ray tracing architecture it is more elegant to implement it via secondary rays in the shader. Rather than generating a set of completely new rays at the first surface, the original ray packet is re-used by specifying a minimum hit distance for each ray. Thus, the secondary packet has exactly the same (common) origin, corner rays and frustum as the primary packet, allowing for all of the aforementioned optimizations. Rays that do not require a transparency ray are disabled, sometimes leading to partially-filled packets, but incurring no additional traversal steps or isopolygon intersections. Note that even though a BVH can have overlapping subtrees, shading will always be performed front-to-back, so both shadows and transparency are always computed accurately (Figure 8.5).

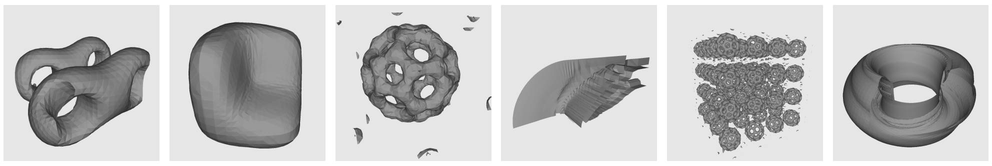

Figure 8.6: From left to right: ell32P (149K tets), feok (122K), bucky ball (177K), bluntfin (225K tets, two isosurfaces), bucky cube (4x4x4 bucky balls, for a total of 11.3m tets), and time step 50 of the fusion data set. With simple shading, these examples run at 14.2, 12.6, 13.3, 18.9, 2.8, and 3.3 frames per second (1024×1024 pixels) on a Intel Core 1 duo laptop with 1GB RAM, and at 95, 93, 90, 94, 19.1, and 26.1 frames per second on a 16-core 2.4 GHz Opteron workstation.

8.7 Results and Discussion

Thus far, performance tradeoffs of individual algorithmic components are addressed in their respective sections. In this section, benchmarks for the system as a whole are considered, and evaluate the overall success of coherent BVH ray tracing for tet-volume isosurfaces. For the experiments, three representative machines are used: a laptop equipped with an Intel Core (1) Duo 2.33 GHz and 1 GB RAM; a Mac Pro desktop PC with a dual Intel Core 2 Duo 2.66 GHz and 4 GB RAM; and a 8-CPU dual-core (16 cores total) Opteron 2.4 GHz workstation with 64 GB RAM. In general the experiments showed that the desktop frequently performed on par with the workstation, except in the case of multiple transparent isosurfaces on large data where the large L2 cache of the workstation had a major impact. If not mentioned otherwise, all examples run at 1024×1024 pixels, and use packets of 16×16 rays. The data sets and scenes used for comparison are depicted in Figures 8.6 and 8.3.

8.7.1 Build Time and Performance

Because a tetrahedral mesh has far less geometric variation than a polygonal model (i.e., tets form a partition of space, and never overlap or self-intersect), the qualitative difference between a SAH and a BIH build is virtually nonexistent (Table 8.1). Because of the lower build times, the BIH-style build is used as default, though the SAH could potentially be useful for extremely irregular data. However, with the fast BIH-style build, most of the smaller data sets could in fact be rebuilt from scratch per frame.

8.7.2 Rendering Performance

As can be seen from Table 8.1 and Table 8.2, all of the static examples can be rendered at multiple frames per second even on the dual-core laptop. For

	Ell32P	Feok	Bucky	Blunt	BuckyCube	Fusion (t=50)
#Tets	148,955	121,668	224,874	176,856	11.3m	3m x 116
Render Performance (Frames per Second)						
BIH	27.0	23.6	18.8	28.4	6.23	11.47
SAH	26.3	23.6	18.9	28.5	6.30	12.13
Build Time (ms, dual Intel Core 2 Duo 2.66 GHz)						
BIH	46	42	61	87	4988	1495
SAH	2432	1854	2932	3887	312620	70689

Table 8.1: BIH-style build vs SAH for building the Implicit BVH. Because the tetrahedra are distributed over space for more evenly than triangles in a polygonal model, the render performance between BIH-style build and SAH build is very similar, but executing the BIH-style build is much faster.

static scenes, performance is typically linear in the number of CPU cores, but with an upper bound of 50–60 fps due to the cost of writing ray-traced pixels to the GPU frame buffer. Empirically, the application scales roughly linearly with respect to number of pixels per frame. Thus, a frame buffer of 512×512 generally renders four times faster than at 1024×1024, allowing for quite interactive rates even when rendering difficult scenes on the laptop.

Scalability in Model Size: Performance degrades quite gracefully when increasing model size, dropping at most by 4x when going from the smallest model (feok, 121k tets) to the most complex one (buckycube, 11.3m tets), even though the latter has nearly 100 times the number of tets. This is largely due to the logarithmic complexity of ray tracing efficiency structures, and the packet-amortized cost of memory access. To further evaluate scalability to large models, several example scenes are generated where a bucky ball is replicated $n \times n \times n$ times *without instancing.* As evident in Table 8.3, performance drops moderately even for hugely complex models of up to nearly a billion tets.

Comparison to Existing Approaches: The results compare quite favorably to the isosurface ray tracing performance achieved by Marmitt et al.'s Plücker-based tet marching algorithm [MS06b], which reported 1.67 and 0.92 fps at 512×512 on a dual-Opteron for isosurfaces on the bluntfin and buckyball, respectively. On comparable hardware (and scaled to same viewport size), the system performs approximately 40 times faster. However, it is important to note that the Marmitt et al. method also supports semi-transparent volume ray casting, which ours doesn't. Comparison with GPU isosurfacing methods is more difficult, due to completely different and continually changing hardware and programming models. Therefore it is restrained

	Ell32P	feok	Bucky	Blunt	BuckyCube	Fusion (t=50)
Laptop	14.2	12.6	13.3	18.9	2.8	3.3
Desktop	29.4	25.9	27.3	45.5	7.21	8.47
Workstation	95	93	90	94	19.1	26.1

Table 8.2: Performance in frames per second for various data sets and platforms: *Laptop* is an Intel Core Duo 2.33 GHz, 1 GB RAM. *Desktop* is a 4-core dual Intel Core 2 Duo 2.66 GHz, 4 GB RAM. *Workstation* is a 16-core cc-NUMA 2.4 GHz Opteron, with 64 GB RAM. Refer to Figure 8.6 for images.

# Replications	1	2^3	4^3	8^3	16^3
# Tets Total	177k	1.4m	11.3m	90.4m	724m
Frames per Second	34	13.5	5.0	1.8	0.66

Table 8.3: Performance in frames per second on four Opteron 2.4GHz cores, for varying numbers of replication of the bucky ball scene (no instancing is used).

from any absolute comparisons, but believe that the frame rates achieved are sufficiently interactive to compete with most GPU based methods for large data sets, while offering more flexibility and unconditional accuracy.

8.7.3 Traversal Efficiency

The key to this interactive performance lies in the aggressive large-packet traversal scheme, as can be seen from Table 8.4. Speculative descent and frustum culling greatly reduce the number of individual ray-box tests during traversal by roughly a factor of 18–51 compared to tracing 2×2 packets (the smallest an SSE-based system can trace). Using packets allows for traversal and intersection code in SSE, which is crucial to realizing the performance potential of modern CPU's.

Because the ray-isotet intersection is transformed to a polygonal problem, the same frustum culling techniques can also be used to significantly reduce the number of individual ray-isopolygon tests, by about 2–3×, even though for the most complex scene the number of ray-isopolygon tests actually increases (see Table 8.4). Finally, larger packets allow for amortizing per-packet operations like isorange culling and isotet extraction over the entire packet, thus reducing the total number of these operations per frame. As evident in Table 8.4, this reduces the number of isopolygon generations by about 6–40×, and the number of culling tests by 22–55×.

# Scene	Bluntfin	Buckyball	Ell32P	Feok	Fusion50	BuckyCube
Number of Individual Ray-Box Tests						
2x2	48.05	93.84	56.75	57.42	175.83	95.89
16x16	0.94	1.8	1.11	1.10	4.32	5.44
Ratio	51×	52×	52×	52×	41×	18×
Number of Individual Ray-Isopolygon Tests						
2x2	8.90	13.52	8.0	12.45	29.35	15.51
16x16	3.19	4.42	3.39	3.86	16.47	23.91
Ratio	2.8×	3.0×	2.4×	3.5×	1.8×	0.65×
Number of Total Packet Isorange Tests						
2x2	76.75	152.31	99.89	95.96	279.75	181.48
16x16	1.45	2.84	1.88	1.79	6.48	8.29
Ratio	51×	54×	53×	53×	43×	22×
Number of Total Isopolygon Extractions (×1000)						
2x2	2216	354	1908	4436	7285	3468
16x16	69	10	6429	109	296	616
Ratio	32×	34×	29×	41×	25×	5.6×

Table 8.4: Traversal statistics of using the aggressive packet-frustum traversal scheme (using 16×16 rays) vs. standard 2×2 packet traversal.

Isopolygon caching vs. On-the-Fly Recomputation: Because the large packets reduce the number of isopolygon extractions, caching the isopolygons has a relatively low impact. Even when using only a single CPU and a large enough cache (so no conflicts occur, and all synchronization can be disabled), caching only increases total frame rate by 5–8% over on-the-fly recomputation, thus on-the-fly recomputation is used by default.

8.7.4 Multiple Isosurfaces, Shadows, and Transparency

As mentioned in Section 8.6, more advanced shading bears a significant cost, mostly due to the higher number of rays traced in the scene. Shadows usually increase the render cost by about 2x if the rendered object covers the entire screen, and somewhat less, otherwise (also see Figure 8.5). Of course, adding more shadow-casting light sources–or even soft shadows or global illumination–would further increase the cost per image, making these effects infeasible on low-end hardware.

Transparency, too, adds to the number of rays traced per image, and correspondingly increases the render cost, particularly if the object has a high depth complexity. For this reason, the number of transparency levels is typically reduced to a user-specified maximum (2 by default), which can be changed interactively. All these effects can be supported simultaneously, even for the complex time-varying data sets (see Figures 8.5 and 8.3).

With diffuse shading, supporting multiple isosurfaces in itself does not significantly raise the cost of an image, due to the ray tracer's implicit occlusion culling (the 2× drop in frame rate in Figure 8.5 is entirely due to the 2× higher projected area of the model after adding the outer isosurface). Adding the clip-box in Figure 8.5 is virtually cost-free.

8.7.5 Time-Varying Data Sets

With isopolygon caching disabled by default, the performance for handling time-varying data depends entirely on the cost of retrieving the BVH and geometry for the proceeding frame. When BVH and vertex positions are precomputed for each frame, switching to a new BVH has no measurable performance impact, as switching requires only changing a few pointers, and models are too large to remain resident in L2 cache anyway. On the other hand, precomputation requires a lavish amount of main memory: for the fusion data set, storing a precomputed BVH and vertices for each time step currently requires a total of 21 GB of memory. Though it can be argued that this could be significantly reduced, this memory footprint is still significant.

Without replicating the vertex arrays and precomputing the BVHs, all 116 time steps of the 3 million tet fusion data set can be fit into 538 MB (including one shared BVH that is refit per frame), allowing us to render even that model on the laptop. However, refitting requires updating the vertex array, all the BVH nodes, and some precomputed shading data (e.g., per-tet gradients) per frame, adding a significant per-frame cost that limits maximum performance. The update is fully parallelized, but – unlike rendering – scales poorly due to intensive and asymmetrical memory access on that particular workstation's cc-NUMA architecture.

In short, precomputation and refitting offer a classical trade-off between performance and memory consumption. For the fusion data set shown in Figure 8.3 with all effects turned on, precomputation results in 7–15 fps on the 16-core Opteron, but requires 21 GB or memory. Refitting requires only 538 MB of memory, but is limited to 3.5 fps when to a new time step is switched every frame.

For smaller models, interactive refitting is not an issue, and for model sizes of 100K–250K tets even per-frame rebuilds are feasible (see Table 8.1). This would even allow for applications where neither scalar field, nor number of tets, nor mesh topology are known in advance. For models as large as the fusion data set, this is currently not possible at interactive rates. However, as the serial BIH build is sufficiently fast that an efficiently implemented distributed build could permit fully dynamic rebuilding.

8.8 Conclusions and Future Work

In this chapter it is shown that it is possible to ray trace isosurfaces of tetrahedral scalar fields at interactive to real-time frame rates, purely on the CPU. In doing so, it is possible to correctly visualize large unstructured volumes, interactively manipulate isovalues and shader modalities, and handle time-varying data with hundreds of steps.

The main algorithmic contributions are a fast packet-isotetrahedron intersection test and extension of the coherent BVH to an implicit min-max tree over the tetrahedral volume. The implementation naturally supports multiple isosurfaces, on-the-fly clipping, semi-transparent depth peeling, and shadows. Accommodation of large data is limited only by host memory capacity, though the overhead of the BVH must be taken into consideration. Time-varying data can be handled by either precomputing an implicit BVH, or by building a single BVH per time step that is updated on the fly. In the former case, one can jump immediately between arbitrary time steps – a feat that would be difficult for streaming GPU methods. Overall, a practical tool for visualizing large tetrahedral data sets is presented.

The approach opens several avenues for future work. For example it should be possible to the extend BVH traversal to direct volume rendering methods, such as maximum intensity projection (MIP) or full transfer-function methods. Though the latter suffer from high traversal complexity, the BVH could still be useful for space-skipping when the transfer function is sufficiently sparse, as in [KW03a]. Another intriguing extension would be support for higher-order finite elements in the spirit of Nelson et al. [NK05] or Rössl et al. [RZNS04]. This would require a completely different intersection routine, but the BVH traversal would remain unchanged. For complicated nonlinear implicit expressions, a robust arbitrary implicit intersector such as Knoll et al. [KHH+07, KHK+07] could be employed. Also of interest would be more advanced lighting effects such as soft shadows, ambient occlusion, or global illumination, which can significantly improve understanding of data sets [Gri06].

Finally, investigating scalable build algorithms could allow for rendering even complex data with arbitrary deformations without precomputation.

CPU ray tracing is practical for visualization applications in the near term. Though GPU's continue to outpace multi-core CPU's in computational power, limited on-board memory and bus latency restrict addressing of large data from the GPU. Fewer such limitations exist for CPU workstations or clusters. Thus, this method is geared primarily towards cutting-edge data sets that are too large for naïve GPU methods, and scientific applications where consistently correct visualization is crucial. In the long run, regardless of hardware platform, isosurface ray tracing, with its logarithmic complexity, is inherently scalable to large data, and trivial to parallelize. As GPU programming models increase in algorithmic flexibility i.e. using CUDA [Buc07], and mainboard memory access improves in transparency, ray tracing methods will likely be ported to graphics hardware, replacing rasterization for large data visualization.

Chapter 9

Final Summary, Future Work, and Final Conclusions

This last chapter summarizes again the main contributions and results of this dissertation and discusses potential future work before final conclusions are drawn.

Final Summary

This dissertation has presented several contributions to broaden the applicability of ray tracing in 3D computer games and isosurface visualization applications. To do so, it concerned itself with questions like: what are the benefits of ray tracing for future computer games, how can certain dynamic scenes efficiently be ray traced, how can ray tracing be implemented efficiently on an IBM-Cell CPU, and how can isosurfaces be ray traced efficiently. In particular, the chapters in this work showed:

Chapter 3 analyzed the benefits that ray tracing based game technology offers, and the current state of ray tracing with regards to 3D computer games. Furthermore, some of the fundamental differences between rasterization driven and ray tracing based games engines were highlighted, as well as prototype ray tracing based games presented.

Chapter 4 presented an approach to efficiently ray trace keyframe as well as skinned animations. The algorithm works by introducing a preprocess that clusters mesh segments with locally coherent motion. For each cluster, or submesh, a fuzzy acceleration structure is build that is valid for the complete animation. Then, for every frame, only a small top-level structure has to be constructed with optimized bounds for the particular time step of the animation. This procedure allows to ray trace dynamic scenes at realtime frame rates with almost no runtime cost. Compared to optimized trees a factor of two in ray tracing performance is lost in average, but ultimately there is a performance net gain because it would be much more expensive to rebuild the acceleration structures from scratch.

Chapter 5 presented an optimized ray tracing architecture for the IBM-Cell processor. The basic implementation uses BVHs, packet/frustum traversal, and an optimized variant of Wald's ray triangle intersection test. In order to achieve fast ray shooting performance, software multithreading as well as manual data caching for triangle and BVH nodes is exploited. The results show that such an approach can achieve on an SPE a similar performance as an optimized ray tracer running on a x86 core. However, a Cell processor has eight compute cores and achieves with them, on a chip-to-chip comparison, higher ray tracing performance than any other current ray tracing implementation.

Chapter 7 described the implicit min/max kD-tree as well as a fast and exact ray isosurface intersection test for volumetric datasets with rectilinear cells. The presented solution clearly outperforms all previously published isosurface ray tracing methods and is at least on par with approaches that have been published recently. It is also shown how the implicit min/max kD-tree approach can be extended to support out-of-core data sets. With this extension is it possible to render gigabyte datasets interactively on todays PCs.

Chapter 8 introduced the implicit min/max BVH with a speculative packet/frustum traversal for isosurface ray tracing of unstructured (dynamic) volumetric datasets. Furthermore, a ray isosurface intersection test was presented that is based on the marching tetrahedra algorithm. Together, this techniques allow to ray trace large time-varying isosurfaces at interactive rates including advanced effects like clipping, transparencies etc.

Future Work

Although it is possible to achieve impressive ray shooting performance with current algorithms on todays CPU and GPUs all has not been said and done. There is always a need and opportunity for more performance.

For example, currently all well known ray tracing implementations that exploit multicore CPUs perform the complete ray tracing pipeline (traversal, intersection, shading) on every single core. This stresses the caches and it might be worthwhile to exploit a vertical distribution of the separate tasks to the processor cores. Furthermore, writing optimized shaders for several different platforms is time consuming and error-prone, e.g. the shaders for the presented Cell implementation do currently not support multithreading and caching. Adding these features manually is just too much work and new high level shading languages and compilers could be used to generate optimized code for different back-ends.

Support for dynamic scenes requires also more research. Currently, all existing solutions are limited in the dynamics they can support, or require complicated strategies to selectively rebuild the acceleration structures which can lead again to too long and unpredictable construction times. A solution might be to use construction methods which are cheap enough such that a full rebuild of the acceleration structure **is** possible for each frame. One possibility could be the use an one dimensional linear ordering of all objects in a scene, e.g. a space-filling curve is used to spatially sort the primitives. An acceleration structure could then be build quickly over the sorted primitives.

Isosurface ray tracing of time-varying out-of-core volumetric datasets is also still a challenge. These datasets are huge and allocate terabytes of hard disk space. Even using e.g. the small presented acceleration structure for rectilinear datasets would result in too much data. For such datasets new concepts are needed which allow to change the isovalue and time step on-the-fly but provide instantly enough information for a meaningful visualization.

Another interesting area for future research are *hybrid algorithms.* Hybrid algorithms first execute a rasterization pass, and then ray tracing will be used e.g. for exact shadow or reflection computations. This would allow to use the best of both worlds and also offer the opportunity to slowly integrate ray tracing into already exiting rasterization based render engines.

Final Conclusions

This dissertation has presented several practical techniques to broaden the applicability of ray tracing for future 3D computer games and isosurface visualization applications. Although rasterization will not be replaced by ray tracing on a massive scale at any time soon, it can be assumed that ray tracing will be more and more important in future. Of course not all problems are solved yet, but new (hybrid)-algorithms, data structures, and massive parallel CPU and GPU architectures will further improve the applicability of ray tracing for real-world rendering applications.

Appendix A

A List of Related Papers

The chapters of this thesis are based on the following publications and the result of collaborative work with the according authors.

Chapter 2:
Exploring the Use of Ray Tracing for Future Games
H. Friedrich, J. Günther, A. Dietrich, M. Scherbaum, H. P. Seidel, and P. Slusallek
Proceedings of ACM SIGGRAPH Video Game Symposium, Boston, USA, 2006

Chapter 3:
Ray Tracing Animated Scenes using Motion Decomposition
J. Günther, H. Friedrich, I. Wald, H. P. Seidel, and P. Slusallek
Proceedings of Eurographics, Vienna, Austria, 2006

Interactive Ray Tracing of Skinned Animations
J. Günther, H. Friedrich, H. P. Seidel, and P. Slusallek
Proceedings of Pacific Graphics, Taipei, Taiwan, 2006

Chapter 4:
Ray Tracing on the CELL processor
C. Benthin, I. Wald, and H. Friedrich
Proceedings of the IEEE Symposium on Interactive Ray Tracing, Salt Lake City, USA, 2006

Chapter 5:
Interactive Volume Rendering with Ray Tracing
G. Marmitt, H. Friedrich, and P. Slusallek
Eurographics State-of-the-Art Report 2006, Vienna, 2006

Chapter 6:
Faster Isosurface Ray Tracing using Implicit KD-Trees
I. Wald, H. Friedrich, G. Marmitt, P. Slusallek, and H. P. Seidel
IEEE Transactions on Visualization and Computer Graphics, 2005

Fast and Accurate Ray-Voxel Intersection Techniques for Iso-Surface Ray Tracing
G. Marmitt, A. Kleer, I. Wald, H. Friedrich, and P. Slusallek
Proceedings of Vision, Modeling, and Visualization, Stanford, USA, 2004

Chapter 7:
Interactive Isosurface Ray Tracing of Time-Varying Tetrahedral Volumes
I. Wald, H. Friedrich, A. Knoll, and C. D. Hansen
Proceedings of IEEE Visualization/InfoVis, Sacramento, USA, 2007

Bibliography

[ADF$^+$] S. Asano, S.H. Dhong, B. Flachs, G. Gervais, A. Hatakeyama, P. Hotstee, R. Kim, T. Le, J. Leenstra, J. Liberty, P. Liu, B. Michael, S.M. Mueller, H. Oh, O. Takahashi, Y. Watanabe, and N. Yano. A streaming processing unit for a cell processor. *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International.*

[Adv03] Advanced Micro Devices. Software Optimization Guide for AMD Athlon(tm) 64 and AMD Opteron(tm) Processors, 2003.

[AM00] Marc Alexa and Wolfgang Müller. Representing animations by principal components. *Computer Graphics Forum*, 19(3):17–24, 2000.

[AMD] Advanced Micro Devices. *Inside 3DNow![tm] Technology.* http://www.amd.com/products/cpg/k623d/inside3d.html.

[AOW06] Junghyun Ahn, Seungwoo Oh, and Kwangyun Wohn. Optimized motion simplification for crowd animation: Research articles. *Comput. Animat. Virtual Worlds*, 17(3):155–165, 2006.

[App68] Arthur Appel. Some Techniques for Shading Machine Renderings of Solids. *Proceedings of the Spring Joint Computer Conference*, pages 27–45, 1968.

[ARFPB90] John M. Airey, John H. Rohlf, and Jr. Frederick P. Brooks. Towards image realism with interactive update rates in complex virtual building environments. In *SI3D '90: Proceedings of the 1990 symposium on Interactive 3D graphics*, pages 41–50, New York, NY, USA, 1990. ACM Press.

[ART03] ARTVPS. Pure PCi-X 3D Rendering Card
. http://www.artvps.com/page/15/pure.htm, 2003.

[AW87] John Amanatides and Andrew Woo. A Fast Voxel Traversal Algorithm for Ray Tracing. In *Proceedings of Eurographics*, pages 3–10. Eurographics Association, 1987.

[Bad90] Didier Badouel. *An efficient ray-polygon intersection.* Academic Press Professional, Inc., San Diego, CA, USA, 1990.

[BEL$^+$07] Solomon Boulos, Dave Edwards, J Dylan Lacewell, Joe Kniss, Jan Kautz, Peter Shirley, and Ingo Wald. Packet-based Whitted and Distribution Ray Tracing. In *Proc. Graphics Interface*, May 2007.

[Ben06] Carsten Benthin. *Realtime Ray Tracing on Current CPU Architectures.* PhD thesis, Computer Graphics Group, Saarland University, 2006.

[Bet05] Bethesda Softworks LLC. The Elder Scrolls IV: Oblivion. http://www.elderscrolls.com/, 2005.

[BPTZ99] C. L. Bajaj, V. Pascucci, D. Thompson, and X. Y. Zhang. Parallel accelerated isocontouring for out-of-core visualization. In *PVGS '99: Proceedings of the 1999 IEEE symposium on Parallel visualization and graphics*, pages 97–104, New York, NY, USA, 1999. ACM Press.

[Bre03] Mark W. Brehob. *On the Mathematics of Caching.* PhD thesis, Michigan State University, 2003.

[Buc07] Ian Buck. GPU Computing with NVIDIA CUDA. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, page 6, New York, NY, USA, 2007. ACM Press.

[BWS03a] Carsten Benthin, Ingo Wald, and Philipp Slusallek. A Scalable Approach to Interactive Global Illumination. *Computer Graphics Forum*, 22(2):621–630, June 2003.

[BWS03b] 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.

[BWS06a] Carsten Benthin, Ingo Wald, and Philipp Slusallek. Techniques for interactive ray tracing of Bézier surfaces. *Journal of Graphics Tools*, 11(2), 2006. (to appear).

[BWS06b] Solomon Boulos, Ingo Wald, and Peter Shirley. Geometric and Arithmetic Culling Methods for Entire Ray Packets. Technical Report UUCS-06-010, SCI Institute, University of Utah, 2006.

[Cat74] Edwin Earl Catmull. *A subdivision algorithm for computer display of curved surfaces.* PhD thesis, 1974.

[CDR02] Alan Chalmers, Timothy Davis, and Erik Reinhard, editors. *Practical Parallel Rendering.* A K Peters, 2002. ISBN 1-56881-179-9.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* The MIT Press, 2nd edition, 2001.

[CMM$^+$97] Paolo Cignoni, Paola Marino, Claudio Montani, Enrico Puppo, and Roberto Scopigno. Speeding Up Isosurface Extraction Using Interval Trees. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):158–170, 1997.

[CPC84] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 137–145, New York, NY, USA, 1984. ACM Press.

[CW93] Michael F. Cohen and John R. Wallace. *Radiosity and Realistic Image Synthesis.* Morgan Kaufmann Publishers, 1993.

[DCDS05] 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.

[DCH88] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 65–74, New York, NY, USA, 1988. ACM Press.

[DGP04] 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.

[DHS04] Kirill Dmitriev, Vlastimil Havran, and Hans-Peter Seidel. Faster Ray Tracing with SIMD Shaft Culling. Research Report MPI-I-2004-4-006, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 2004.

[DK91a] A. Dai and A. Koide. An Efficient Method of Triangulating Equi-Valued Surfaces by Using Tetrahedral Cells. In *IEICE Trans. Commun. Elec. Inf. Syst.*, pages 214–224, 1991.

[DK91b] Akio Doi and Akoi Koide. An efficient method of triangulating equi-valued surfaces by using tetrahedral cells. *IEICE Trans Commun. Elec. Inf. Syst*, E-74(1):213–224, 1991.

[DPH$^+$03] 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.

[DS06] Carsten Dachsbacher and Marc Stamminger. Splatting indirect illumination. In *SI3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 93–100, New York, NY, USA, 2006. ACM Press.

[Dur99] Frédo Durand. *3D Visibility: Analytical Study and Applications.* PhD thesis, Université Joseph Fourier, Grenoble I, July 1999. http://www-imagis.imag.fr.

[DWBS03] 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.

[DWS05] Andreas Dietrich, Ingo Wald, and Philipp Slusallek. Large-Scale CAD Model Visualization on a Scalable Shared-Memory Architecture. In Günther Greiner, Joachim Hornegger, Heinrich Niemann, and Marc Stamminger, editors, *Proceedings of 10th International Fall Workshop - Vision, Modeling, and Visualization (VMV) 2005*, pages 303–310, Erlangen, Germany, November 2005. Akademische Verlagsgesellschaft Aka.

[EAMJ05] Manfred Ernst, Tomas Akenine-Möller, and Henrik Wann Jensen. Interactive rendering of caustics using interpolated warped volumes. In *GI '05: Proceedings of the 2005 conference on Graphics interface*, pages 87–96, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2005. Canadian Human-Computer Communications Society.

[EKE01] Klaus Engel, Martin Kraus, and Thomas Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 9–16, 2001.

[FKN80] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On visible surface generation by a priori tree structures. In *SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, pages 124–133. ACM Press, 1980.

[FS05] Tim Foley and Jeremy Sugerman. KD-tree Acceleration Structures for a GPU Raytracer. In *HWWS '05 Proceedings*, pages 15–22, New York, NY, USA, 2005. ACM Press.

[FSY$^+$06] Tim Foley, Jeremy Sugerman, Shigeatsu Yoshioka, , and Pat Hanrahan. Ray Tracing on a Cell Processor with Software Caching. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, page 8, 2006.

[FvDFH97] Foley, van Dam, Feiner, and Hughes. *Computer Graphics – Principles and Practice, 2nd edition.* Addison Wesley, 1997.

[GA05] Markus Geimer and O. Abert. Interactive Ray Tracing of Trimmed Bicubic Bezier Surfaces without Triangulation. *WSCG'2005 Full Papers Conference Proceedings*, pages 71–78, 2005.

[GBKG04] Sören Grimm, Stefan Bruckner, Armin Kanitsar, and Meister Eduard Gröller. Memory efficient acceleration structures and techniques for cpu-based volume raycasting of large data. In *Proceedings IEEE/SIGGRAPH Symposium on Volume Visualization and Graphics*, pages 1–8, October 2004.

[GFSS06] Johannes Günther, Heiko Friedrich, Hans-Peter Seidel, and Philipp Slusallek. Interactive ray tracing of skinned animations. *The Visual Computer*, 22(9):785–792, September 2006. (Proceedings of Pacific Graphics).

[GFW$^+$06] Johannes Günther, Heiko Friedrich, Ingo Wald, Hans-Peter Seidel, and Philipp Slusallek. Ray tracing animated scenes using motion decomposition. *Computer Graphics Forum*, 25(3):517–525, September 2006. (Proceedings of Eurographics).

[Gla89] Andrew Glassner. *An Introduction to Ray Tracing.* Morgan Kaufmann, 1989.

[Gla90] Andres Glassner, editor. *Graphics Gems.* Academic Press, 1990.

[GPSS07] Johannes Günther, Stefan Popov, Hans-Peter Seidel, and Philipp Slusallek. Realtime ray tracing on GPU with BVH-based packet traversal. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007*, September 2007.

[Gri05] Sören Grimm. *Real-Time Mono- and Multi-Volume Rendering of Large Medical Datasets on Standard PC Hardware.* PhD thesis, Technischen Universität Wien, 2005.

[Gri06] Christiaan Gribble. *Interactive Methods for Effective Particle Visualization.* PhD thesis, University of Utah, 2006.

[GS87] Jeffrey Goldsmith and John Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, 7(5):14–20, May 1987.

[GWS04] Johannes Günther, Ingo Wald, and Philipp Slusallek. Realtime Caustics using Distributed Photon Mapping. In *Rendering Techniques 2004*, pages 111–121, June 2004.

[Hav01a] Vlastimil Havran. *Heuristic Ray Shooting Algorithms.* PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, 2001.

[Hav01b] Vlastimil Havran. *Heuristic Ray Shooting Algorithms.* PhD thesis, Czech Technical University in Prague, 2001.

[Hil89] W. Danny Hillis. *The Connection Machine.* MIT-Press, 1989.

[HJ04] Charles Hansen and Chris R. Johnson. *The Visualization Handbook.* Elsevier, 2004.

[HL79] G.T. Hermann and H.K. Lin. Three dimensional display of human organs from computed tomograms. In *Computer Vision, Graphics and Image Processing*, pages 1–21, 1979.

[HMS06] Warren Hunt, William R. Mark, and Gordon Stoll. Fast kd-tree construction with an adaptive error-bounded heuristic. In *2006 IEEE Symposium on Interactive Ray Tracing*. IEEE, Sept. 2006.

[Hur05] Jim Hurley. Ray tracing goes mainstream. *Intel Technology Journal*, 9(2):99–108, 2005.

[idS99] idSoftware.
Quake III Arena.
http://www.idsoftware.com/games/quake/quake3-arena/,
1999.

[idS06] idSoftware.
Quake IV.
http://www.idsoftware.com/games/quake/quake4/, 2006.

[IKW07] Thiago Ize, Andrew Kensler, and Ingo Wald. A coherent grid traversal approach to visualizing particle-based simulation data. *IEEE Transactions on Visualization and Computer Graphics*, 13(4):758–768, 2007. Member-Christiaan P. Gribble and Member-Steven G. Parker.

[Int02a] Intel Corp.
Intel C/C++ Compilers, 2002.
http://www.intel.com/software/products/compilers.

[Int02b] Intel Corp.
Intel Pentium III Streaming SIMD Extensions.
http://developer.intel.com/vtune/cbts/simd.htm, 2002.

[Int02c] Intel Corp.
Introduction to Hyper-Threading Technology.
http://developer.intel.com/technology/hyperthread, 2002.

[Int05] International Business Machines. The Cell Project at IBM Research. http://www.research.ibm.com/cell/, 2005.

[inT06] inTrace. Company Hompage. http://www.intrace.com/, 2006.

[ISP07] Thiago Ize, Peter Shirley, and Steven G. Parker. Grid creation strategies for efficient ray tracing. In *2007 IEEE Symposium on Interactive Ray Tracing*. IEEE, Sept. 2007.

[IWP07] Thiago Ize, Ingo Wald, and Steven G Parker. Asynchronous BVH Construction for Ray Tracing Dynamic Scenes on Parallel Multi-Core Architectures. In *Proceedings of the 2007 Eurographics Symposium on Parallel Graphics and Visualization*, 2007.

[Jan86] F. W. Jansen. Data structures for ray tracing. In *Proceedings of the workshop on Data structures for Raster Graphics*, pages 57–73, 1986.

[JD88] Anil K. Jain and Richard C. Dubes. *Algorithms for Clustering Data.* Prentice Hall Advanced Reference Series: Computer Science, 1988.

[Jen01] Henrik Wann Jensen. *Realistic Image Synthesis Using Photon Mapping.* A K Peters, 2001.

[Kaj86] James T. Kajiya. The rendering equation. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 143–150, New York, NY, USA, 1986. ACM Press.

[Kap85] Michael R. Kaplan. Space Tracing: A Constant Time Ray Tracer. In *SIGGRAPH 85 Tutorial on the State of the Art in Image Synthesis*, July 1985.

[KDH$^+$05a] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell Multiprocessor. *IBM Journal of Research and Development*, 49(4-5):589–604, 2005.

[KDH$^+$05b] James A. Kahle, Michael N. Day, H. Peter Hofstee, Charles R. Johns, Theodor R. Maeurer, and David Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4):589–604, 2005.

[KDK$^+$01] Brucek Khailany, William J. Dally, Ujval J. Kapasi, Peter Mattson, Jinyung Namkoong, John D. Owens, Brian Towles, Andrew Chang, and Scott Rixner. Imagine: Media processing with streams. *IEEE Micro*, 21(2):35–46, 2001.

[Kel97] Alexander Keller. Instant Radiosity. *Computer Graphics (Proceedings of ACM SIGGRAPH)*, pages 49–56, 1997.

[Kep75] E. Keppel. Approximating Complex Surfaces by Triangulation of Contour Lines. In *IBM Journal of Research and Development*, pages 2–11, 1975.

[KG04] Z. Karni and C. Gotsman. Compression of soft-body animation sequences, 2004.

[KH95] 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.

[KHH$^+$07] Aaron Knoll, Younis Hijazi, Charles D Hansen, Ingo Wald, and Hans Hagen. Interactive Ray Tracing of Arbitrary Implicit Functions. In *Proceedings of the 2007 Eurographics/IEEE Symposium on Interactive Ray Tracing*, 2007.

[KHK$^+$07] Aaron Knoll, Younis Hijazi, Andrew Kensler, Mathias Schott, Charles Hansen, and Hans Hagen. Fast and Robust Ray Tracing of General Implicits on the GPU. Technical report, 2007.

[KHW07] Aaron Knoll, Charles Hansen, and Ingo Wald. Coherent Multiresolution Isosurface Ray Tracing. Technical Report UUSCI-2007-001, University of Utah, School of Computing, 2007.

[KNT05] Jacob Kogan, Charles Nicholas, and Marc Teboulle. *Grouping Multidimensional Data. Recent Advances in Clustering.* Springer, 2005.

[KSS02] Jan Kautz, Peter-Pike Sloan, and John Snyder. Fast, arbitrary brdf shading for low-frequency lighting using spherical harmonics. In *EGRW '02: Proceedings of the 13th Eurographics workshop on Rendering*, pages 291–296, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[KW03a] Jens Krueger and Ruediger Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings IEEE Visualization 2003*, 2003.

[KW03b] J. Kruger and R. Westermann. Acceleration techniques for GPU-based volume rendering. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 38, Washington, DC, USA, 2003. IEEE Computer Society.

[KWPH06] Aaron Knoll, Ingo Wald, Steven G Parker, and Charles D Hansen. Interactive Isosurface Ray Tracing of Large Octree Volumes. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, 2006.

[LAM01] Jonas Lext and Tomas Akenine-Möller. Towards Rapid Reconstruction for Animated Ray Tracing. In *Eurographics 2001 – Short Presentations*, pages 311–318, 2001.

[LCN98] Barthold Lichtenbelt, Randy Crane, and Shaz Naqvi. *Introduction to Volume Rendering.* Person Education, 1998.

[Lev88] Marc Levoy. Display of surfaces from volume data. *IEEE Comput. Graph. Appl.*, 8(3):29–37, 1988.

[LG95] David Luebke and Chris Georges. Portals and mirrors: simple, fast evaluation of potentially visible sets. In *SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 105–ff., New York, NY, USA, 1995. ACM Press.

[LH98] Yarden Livnat and Charles Hansen. View dependent isosurface extraction. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 175–180, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[Llo82] Stuart P. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28:129–137, 1982.

[LSJ96] Yarden Livnat, Han-Wei Shen, and Christopher R. Johnson. A Near Optimal Isosurface Extraction Algorithm Using the Span Space. *IEEE Transactions on Visualization and Computer Graphics*, 2(1):73–84, 1996.

[LWC$^+$02] David Luebke, Benjamin Watson, Jonathan D. Cohen, Martin Reddy, and Amitabh Varshney. *Level of Detail for 3D Graphics.* Elsevier Science Inc., New York, NY, USA, 2002.

[LYTM06] C. Lauterbach, S.-E. Yoon, D. Tuft, and Manocha. Rt-deform: Interactive ray tracing of dynamic scenes using bvhs. In *2006 IEEE Symposium on Interactive Ray Tracing*, pages 39–45. IEEE, Sept. 2006.

[Mah05] Jeffrey Mahovsky. *Ray Tracing with Reduced-Precision Bounding Volume Hierarchies.* PhD thesis, University of Calgary, 2005.

[Max95] Nelson Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, 1995.

[MB90] David J. MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. *Vis. Comput.*, 6(3):153–166, 1990.

[MCC$^+$99] Arthur A. Mirin, Ron H. Cohen, Bruce C. Curtis, William P. Dannevik, Andris, M. Dimits, Mark A. Duchaineau, D. E. Eliason, Daniel R. Schikore, S. E. Anderson, D. H. Porter, and Paul R. Woodward. Very High Resolution Simulation Of Compressible Turbulence On The IBM-SP System. In *Proceedings of SuperComputing*, 1999. (Also available as Lawrence Livermore National Laboratory technical report UCRL-MI-134237).

[MFK$^+$04] Gerd Marmitt, Heiko Friedrich, Andreas Kleer, Ingo Wald, and Philipp Slusallek. Fast and Accurate Ray-Voxel Intersection Techniques for Iso-Surface Ray Tracing. In *Proceedings of Vision, Modeling, and Visualization (VMV)*, pages 429–435, 2004.

[MFS06] Gerd Marmitt, Heiko Friedrich, and Philipp Slusallek. Interactive Volume Rendering with Ray Tracing. In *Eurographics State of the Art Reports*, 2006.

[MFT05] B. Minor, G. Fossum, and V. To. TRE : Cell Broadband Optimized Real-Time Ray-Caster. In *Proceedings of GPSx*, 2005.

[Mic06] Microsoft. DirectX 9.0. http://www.microsoft.com/-windows/directx/, 2006.

[MMAM07] Erik Mansson, Jacob Munkberg, and Tomas Akenine-Moller. Deep coherent ray tracing. In *2007 IEEE Symposium on Interactive Ray Tracing*, pages 79–85. IEEE, Sept. 2007.

[MMMY97] Torsten Möller, Raghu Machiraju, Klaus Mueller, and Roni Yagel. A comparison of normal estimation schemes. In *VIS '97: Proceedings of the 8th conference on Visualization '97*, pages 19–ff., Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.

[MMU05] Thali M.J., Braun M., and Buck U. VIRTOPSY - Scientific Documentation, Reconstruction and Animation in Forensic: Individual and Real 3d Data Based Geometric Approach including

Optical Body/Object. *Journal of Forensic Sciences*, 50(2):15, 2005.

[MNM06] B. Minor, M. Nutter, and J. Madruga. iRT : An Interactive Ray Tracer for the CELL Processor. In *IBM Techreports*, 2006.

[MS06a] Gerd Marmitt and Philipp Slusallek. Fast ray traversal of tetrahedral and hexahedral meshes for direct volume rendering. In *Proceedings of Eurographics/IEEE-VGTC Symposium on Visualization (EuroVIS) 2006*, pages 235–242, May 2006.

[MS06b] Gerd Marmitt and Philipp Slusallek. Fast Ray Traversal of Tetrahedral and Hexahedral Meshes for Direct Volume Rendering. In *EuroVis 2006*, 2006. (to appear).

[MT97] Tomas Möller and Ben Trumbore. Fast, Minimum Storage Ray Triangle Intersection. *Journal of Graphics Tools*, 2(1):21–28, 1997.

[Muu95] Michael J. Muuss. Towards Real-Time Ray-Tracing of Combinatorial Solid Geometric Models. In *Proceedings of BRL-CAD Symposium*, 1995.

[NFLM07] Paul Arthur Navratil, Donald S. Fussell, Calvin Lin, and William R. Mark. Dynamic ray scheduling to improve ray coherence and bandwidth utilization. In *2007 IEEE Symposium on Interactive Ray Tracing*, pages 95–104. IEEE, Sept. 2007.

[NH90] Gregory M. Nielson and Bernd Hamann. Techniques for the interactive visualization of volumetric data. In *VIS '90: Proceedings of the 1st conference on Visualization '90*, pages 45–50, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[NH91] Greg Nielson and Bernd Hamann. The Asymptotic Decider: Removing the Ambiguity in Marching Cubes. In G. Nielson and L. Rosenblum, editors, *Proceedings of Visualization '91*, pages 83–91. IEEE Computer Society Press, 1991.

[NK05] Blake Nelson and Robert M. Kirby. Ray-tracing polymorphic multi-domain spectral/hp elements for isosurface rendering. *IEEE Transactions on Visualization and Computer Graphics (Proceedings IEEE Visualization 2005)*, 12(1):114–125, 2005.

[NMHW02] A. Neubauer, L. Mroz, H. Hauser, and R. Wegenkittl. Cell-based first-hit ray casting. In *Proceedings of the Symposium on Data Visualisation 2002*, pages 77–ff, 2002.

[Ope06] Open Source Community. The CAL3D Library. https://gna.org/projects/cal3d/, 2006.

[PBMH02] 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.

[PGSS06] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Experiences with streaming construction of SAH KD-trees. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 89–94, September 2006.

[PGSS07] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Stackless kd-tree traversal for high performance gpu ray tracing. *Computer Graphics Forum*, 26(3), September 2007. (Proceedings of Eurographics), to appear.

[PH04] Matt Pharr and Greg Humphreys. *Physically Based Rendering : From Theory to Implementation.* Morgan Kaufman, 2004.

[Pho75] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, 1975.

[PKGH97] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Rendering Complex Scenes with Memory-Coherent Ray Tracing. *Computer Graphics*, 31(Annual Conference Series):101–108, August 1997.

[PMS$^+$99] Steven Parker, William Martin, Peter-Pike J. Sloan, Peter Shirley, Brian Smits, and Charles Hansen. Interactive ray tracing. In *I3D '99: Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 119–126, New York, NY, USA, 1999. ACM Press.

[PPL$^+$99] 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.

[PSL$^+$98] 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.

[Pur04] Timothy J. Purcell. *Ray Tracing on a Stream Processor.* PhD thesis, Stanford University, 2004.

[Res06] Alexander Reshetov. Omnidirectional Ray Tracing Traversal Algorithm for kd-trees. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 57–60, 2006.

[Roc02] Rockstar Games. Grand Theft Auto: Vice City. http://www.rockstargames.com/vicecity/, 2002.

[RSEB$^+$00] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive Volume Rendering on Standard PC Graphics Hardware using Multi-textures and Multi-stage Rasterization. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 109–118. ACM Press, 2000.

[RSH00] Erik Reinhard, Brian Smits, and Chuck Hansen. Dynamic Acceleration Structures for Interactive Ray Tracing. In *Proceedings of the Eurographics Workshop on Rendering*, pages 299–306, Brno, Czech Republic, June 2000.

[RSH05] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-Level Ray Tracing Algorithm. *ACM Transaction of Graphics*, 24(3):1176–1185, 2005. (Proceedings of ACM SIGGRAPH).

[RW80] Steve M. Rubin and Turner Whitted. A three-dimensional representation for fast rendering of complex scenes. *Computer Graphics*, 14(3):110–116, July 1980.

[RZNS04] Christian Rössl, Frank Zeilfelder, Günther Nürnberger, and Hans-Peter Seidel. Reconstruction of Volume Data with Quadratic Super Splines. *IEEE Transactions on Visualization and Computer Graphics*, 10(4):397–409, 2004.

[Sab88] Paolo Sabella. A rendering algorithm for visualizing 3d scalar fields. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 51–58, New York, NY, USA, 1988. ACM Press.

[SDP$^+$04] Jörg Schmittler, Tim Dahmen, Daniel Pohl, Christian Vogelgesang, and Philipp Slusallek. Ray Tracing for Current and Future Games. In *Proceedings of 34. Jahrestagung der Gesellschaft für Informatik*, 2004.

[Seg07] Sega. http://www.sega.com/, 2007.

[SH92] Robert Siegel and John R. Howell. *Thermal Radiation Heat Transfer, Third Edition.* Taylor & Francis, 1992.

[Shi02] Peter Shirley. *Fundamentals of Computer Graphics.* A K Peters, 2002.

[SKS02] Peter-Pike Sloan, Jan Kautz, and John Snyder. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 527–536, New York, NY, USA, 2002. ACM Press.

[SLS05] Peter-Pike Sloan, Ben Luna, and John Snyder. Local, deformable precomputed radiance transfer. *ACM Trans. Graph.*, 24(3):1216–1224, 2005.

[SM00] Heidrun Schumann and Wolfgang Mueller. *Visualisierung - Grundlagen und allgemeine Methoden.* Springer, 2000.

[SM03] Peter Shirley and R. Keith Morley. *Realistic Ray Tracing.* A K Peters, second edition, 2003.

[SSM$^+$05] Peter Shirley, Philipp Slusallek, Bill Mark, Gordon Stoll, and Ingo Wald. Introduction to real-time ray tracing. In *Course notes #38 for ACM SIGGRAPH.* ACM Press, 2005.

[Str03] Gilbert Strang. *Introduction to Linear Algebra.* Wellesley-Cambridge Press, third edition, 2003.

[SWS02] 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.

[SWW$^+$04] 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.

[The01] Holger Theisel. CAGD and Scientific Visualization, Habilitation Thesis, 2001.

[THQ01] THQ Inc. Red Faction. http://www.redfaction.com/, 2001.

[UK88] Craig Upson and Michael Keeler. V-buffer: visible volume rendering. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 59–64, New York, NY, USA, 1988. ACM Press.

[Wal04] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination.* PhD thesis, Computer Graphics Group, Saarland University, 2004.

[Wal07] Ingo Wald. On fast Construction of SAH-based Bounding Volume Hierarchies. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pages 51–58, 2007.

[WBS02] 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.

[WBS03] 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.

[WBS07] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph.*, 26(1):6, 2007.

[WCA$^+$90] Jane Wihelms, Judy Challinger, Naim Alper, Shankar Ramamoorthy, and Arsi Vaziri. Direct volume rendering of curvilinear volumes. In *VVS '90: Proceedings of the 1990 workshop on Volume visualization*, pages 41–47, New York, NY, USA, 1990. ACM Press.

[WDS04] Ingo Wald, Andreas Dietrich, and Philipp Slusallek. An Interactive Out-of-Core Rendering Framework for Visualizing Massively Complex Models. In *Rendering Techniques 2004, Proceedings of the Eurographics Symposium on Rendering*, pages 81–92, 2004.

[WFM$^+$05] Ingo Wald, Heiko Friedrich, Gerd Marmitt, Philipp Slusallek, and Hans-Peter Seidel. Faster Isosurface Ray Tracing using Implicit KD-Trees. *IEEE Transactions on Visualization and Computer Graphics*, 11(5):562–573, 2005.

[WG92] Jane Wilhelms and Allen Van Gelder. Octrees for faster isosurface generation. *ACM Transaction on Graphics*, 11, 1992.

[WH06] Ingo Wald and Vlastimil Havran. On building fast kd-trees for ray tracing, and on doing that in O(N log N). Technical report, SCI Institute, University of Utah, 2006. (submitted for publication).

[Whi80] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, 1980.

[WIK$^+$06] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G. Parker. Ray tracing animated scenes using coherent grid traversal. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 485–493, New York, NY, USA, 2006. ACM Press.

[WK06] Carsten Wächter and Alexander Keller. Instant Ray Tracing: The Bounding Interval Hierarchy. In *Rendering Techniques 2006 – Proceedings of the 17th Eurographics Symposium on Rendering*, pages 139–149, 2006.

[WKB$^+$02] Ingo Wald, Thomas Kollig, Carsten Benthin, Alexander Keller, and Philipp Slusallek. 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).

[WKE99] Rüdiger Westermann, Leif Kobbelt, and Tom Ertl. Real-time Exploration of Regular Volume Data by Adaptive Reconstruction of Iso-Surfaces. *The Visual Computer*, 15(2):100–111, 1999.

[WMG$^+$] Ingo Wald, William R Mark, Johannes Günther, Solomon Boulos, Thiago Ize, Warren Hunt, Steven G Parker, and Peter Shirley. State of the Art in Ray Tracing Animated Scenes. In *Eurographics 2007 State of the Art Reports.*

[WMS98] Peter L. Williams, Nelson L. Max, and Clifford M. Stein. A High Accuracy Volume Renderer for Unstructured Data. *IEEE*

Transactions on Visualization and Computer Graphics, 4(1):37–54, 1998.

[WMS06] Sven Woop, Gerd Marmitt, and Philipp Slusallek. B-kd trees for hardware accelerated ray tracing of dynamic scenes. Technical report, Computer Graphics Group, Saarland University, 2006. (submitted for publication).

[WNDS01] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide, Version 1.2*. Addison Wesley, third edition, May 2001.

[WR02] Bradford J Wood and Pouneh Razavi. Virtual Endoscopy: A Promising new Technology. *American family physician.*, 66, 2002.

[WS01] Ruediger Westermann and Bernd Sevenich. Accelerated Volume Ray-Casting using Texture Mapping. In *IEEE Visualization 2001*, 2001.

[WS05] Ingo Wald and Hans-Peter Seidel. Interactive Ray Tracing of Point Based Models. In *Proceedings of 2005 Symposium on Point Based Graphics (PGB)*, page to appear, 2005.

[WSB01] 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.

[WSBW01] 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).

[WSO$^+$06] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. The potential of the cell processor for scientific computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 9–20, New York, NY, USA, 2006. ACM Press.

[WSS05] Sven Woop, Joerg Schmittler, and Philipp Slusallek. RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. *Proceedings of ACM SIGGRAPH*, 2005.

[WTL$^+$04] Xi Wang, Xin Tong, Stephen Lin, Shi-Min Hu, Baining Guo, and Heung-Yeung Shum. Generalized displacement maps. In Alexander Keller and Henrik Wann Jensen, editors, *Rendering Techniques 2004*, pages 227–234. Eurographics Association, June 2004.

[WV] J Wilhelms and A Van Gelder. Octrees for faster isosurface generation. pages 201–227.

[Wym05] Chris Wyman. An approximate image-space approach for interactive refraction. *ACM Trans. Graph.*, 24(3):1050–1053, 2005.

[YCK92] R. Yagel, D. Cohen, and A. Kaufman. Normal estimation in 3D discrete space. *The Visual Computer*, 8(5-6):278–291, 1992.

[YCM07] Sung-Eui Yoon, Sean Curtis, and Dinesh Manocha. Ray tracing dynamic scenes using selective restructuring. In *SIGGRAPH '07: ACM SIGGRAPH 2007 sketches*, page 55, New York, NY, USA, 2007. ACM Press.

[ZN03] Huijuan Zhang and Timothy S. Newman. Efficient parallel out-of-core isosurface extraction. In *PVG '03: Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, page 3, Washington, DC, USA, 2003. IEEE Computer Society.