Ray Tracing of Dynamic Scenes

Johannes Günther

Saarbrücken, 2014

Dissertation

---

zur Erlangung des Grades
„Doktor der Ingenieurwissenschaften“
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

Tag des Kolloquiums	24.10.2014
Dekan der Naturwissenschaftlich-Technischen Fakultät I	Prof. Dr. Markus Bläser
Vorsitzender des Prüfungsausschusses	Prof. Dr. Jan Reineke
Berichterstatter	Prof. Dr. Philipp Slusallek
	Prof. Dr. Hans-Peter Seidel
	Prof. Dr. Marc Stamminger
Akademischer Beisitzer	Dr. Tobias Ritschel

„Es ist der stetig fortgesetzte, nie erlahmende Kampf gegen Skeptizismus und Dogmatismus, gegen Unglaube und gegen Aberglaube, den Religion und Naturwissenschaft gemeinsam führen, und das richtungsweisende Losungswort in diesem Kampf lautet von jeher und in alle Zukunft: Hin zu Gott!“

Max Planck

Abstract

In the last decade ray tracing performance reached interactive frame rates for non-trivial scenes, which roused the desire to also ray trace *dynamic* scenes. Changing the geometry of a scene, however, invalidates the precomputed auxiliary data-structures needed to accelerate ray tracing. In this thesis we review and discuss several approaches to deal with the challenge of ray tracing dynamic scenes.

In particular we present the *motion decomposition* approach that avoids the invalidation of acceleration structures due to changing geometry. To this end, the animated scene is analyzed in a preprocessing step to split it into coherently moving parts. Because the relative movement of the primitives within each part is small it can be handled by special, pre-built kd-trees. Motion decomposition enables ray tracing of predefined animations and skinned meshed at interactive frame rates.

Our second main contribution is the *streamed binning* approach. It approximates the evaluation of the cost function that governs the construction of optimized kd-trees and BVHs. As a result, construction speed especially for BVHs can be increased by one order of magnitude while still maintaining their high quality for ray tracing.

Kurzfassung

Im letzten Jahrzehnt wurden interaktive Bildwiederholraten bei dem Raytracen von nicht trivialen Szenen erreicht. Dies hat den Wunsch geweckt, auch sich verändernde Szenen mit Raytracing darstellen zu können. Allerdings werden die vorberechneten Datenstrukturen, welche für die Beschleunigung von Raytracing gebraucht werden, durch Veränderungen an der Geometrie einer Szene unbrauchbar gemacht. In dieser Dissertation untersuchen und diskutieren wir mehrere Lösungsansätze für das Problem der Darstellung von sich verändernden Szenen mittels Raytracings.

Insbesondere stellen wir den *Motion Decomposition* Ansatz vor, welcher die bisher nötige Neuberechnung der Beschleunigungsdatenstrukturen aufgrund von Geometrieänderungen zu einem großen Teil vermeidet. Dazu wird in einem Vorberechnungsschritt die animierte Szene untersucht und diese in sich ähnlich bewegende Teile zerlegt. Da dadurch die relative Bewegung der Primitiven der Teilszenen zueinander sehr klein ist kann sie durch spezielle, vorberechnete kd-Bäume toleriert werden. Motion Decomposition ermöglicht das Raytracen von vordefinierte Animationen und Skinned Meshes mit interaktiven Bildwiederholraten.

Unser zweiten Hauptbeitrag ist der *Streamed Binning* Ansatz. Dabei wird die Kostenfunktion, welche die Konstruktion von für Raytracing optimierten kd-Bäumen und BVHs steuert, näherungsweise ausgewertet, wobei deren Qualität kaum beeinträchtigt wird. Im Ergebnis wird insbesondere die Zeit für den Aufbau von BVHs um eine Größenordnung reduziert.

Zusammenfassung

Die Fortschritte des letzten Jahrzehnts in der Geschwindigkeit von Raytracing haben – mit dem Erreichen interaktive Bildwiederholraten für nicht triviale Szenen – den Wunsch geweckt, auch sich verändernde Szenen mittels Raytracings darstellen zu können. Sich verändernde Szenen stellen eine große Herausforderung für Raytracing dar, denn die vorberechneten Datenstrukturen für die Beschleunigung des Bildsyntheseverfahrens werden durch Veränderungen in der Geometrie unbrauchbar gemacht. In dieser Dissertation diskutieren wir mehrere Lösungsansätze für das Problem der Darstellung von sich verändernden Szenen mittels Raytracings. Als Hauptbeiträge entwickelten wir zwei dieser Ansätze: *Motion Decomposition* und *Streamed Binning*.

Das Ziel des Motion Decomposition Ansatzes ist es, möglichst zu vermeiden, dass Veränderungen der Szene eine Neuberechnung der Beschleunigungsdatenstrukturen nach sich ziehen, weil sie ungültig geworden waren. Die Idee des Ansatzes besteht darin, die Szene in Teile zu zerlegen, und damit auch die Bewegungen in der Szene zu zerlegen: in einen Hauptanteil, welcher die Bewegungen der Szenenteile relativ zueinander beschreibt, und einen Restanteil an Bewegung, welcher sich innerhalb eines jeden Szenenteils abspielt. Die Szenenteilstücke werden in einem kleinen kd-Baum auf oberster Hierarchieebene indexiert, welcher schnell für jedes Bild neu aufgebaut werden kann, um die groben, relativen Bewegungen zu berücksichtigen. Die Restbewegungen werden von *toleranten* kd-Bäumen abgedeckt, welche robust gegenüber kleinen Änderungen der Geometrie sind und somit gültig bleiben und keine Neuberechnung benötigen. Der Motion Decomposition Ansatz beruht auf der Annahme, dass Bewegungen räumlich kohärent stattfinden. Zusätzlich muss das Ausmaß der Bewegungen vorher bekannt sein.

Wir zeigen, wie der Ansatz auf zwei Gebieten angewandt wird: auf vordefinierte Animationen und auf Skinned Meshes. Um die Zerteilung der Szene bei vordefinierten Animationen zu finden entwickelten wir ein Zerlegungsverfahren, welches auf Lloyds Relaxation basiert. Es minimiert die Restbewegungen in jedem erzeugten Szenenteil und optimiert so die Geschwindigkeit von Raytracing. Bei Skinned Meshes nutzt das Zerlegungsverfahren die Informationen der Knochentransformationen aus, welche die Deformation des Oberflächennetzes bestimmen. Im Ergebnis kann man mit dem Mo-

tion Decomposition Ansatz diese Arten der veränderlichen Szenen mittels Raytracings mit interaktiven Bildwiederholraten darstellen.

Das Ziel des Streamed Binning Ansatzes ist es, die Zeit zum Erstellen der Beschleunigungsdatenstrukturen zu verringern, so dass diese nach Veränderungen der Szene schnell genug neuberechnet werden können – idealerweise mehrmals in einer Sekunde.

Als Grundlage beschreiben und analysieren wir die konventionelle Methode für den Aufbau von Beschleunigungsdatenstrukturen, welche mit der Surface-Area Heuristik (SAH) für Raytracing optimiert werden. Daraus leiten wir den Streamed Binning Ansatz ab, welcher sowohl den Aufbau von kd-Bäumen als auch den Aufbau von Bounding Volume Hierarchies (BVHs) beschleunigt. Dazu wird das Minimum der SAH Kostenfunktion und somit die beste Position für die Teilung eines Baumknotens näherungsweise bestimmt, indem die Anzahl der untersuchten Positionen deutlich reduziert wird. Zusätzlich vermeiden wir die vorher zu Beginn des Aufbaus notwendige Sortierung der Geometrieprimitiven, was zu schnellen, zusammenhängenden Speicherzugriffen führt.

Der eingeführte Näherungsfehler ist sehr klein, so dass die erzeugten kd-Bäume und BVHs von hoher Qualität für Raytracing sind. Daher kann der Streamed Binning Ansatz den konventionellen Aufbau der Beschleunigungsdatenstrukturen in den meisten Einsatzgebieten ersetzen. Wir zeigen, dass der Aufbau von BVHs deutlich mehr von unserer neuen Methode profitiert als der Aufbau von kd-Bäumen. BVHs können bereits schnell genug neu aufgebaut werden, um beliebige, interaktive Veränderungen an Szenen von kleinerer bis mittlerer Größe zu unterstützen. Bei sehr großen Szenen verringert der Streamed Binning Ansatz die Zeit bis zum Erscheinen des ersten Bildes um eine Größenordnung.

Wie wir in dieser Dissertation zeigen gibt es mehrere unterschiedliche Ansätze, um sich verändernde Szenen mittels Raytracings darzustellen, welche jeweils für verschiedene Gebiete und Anforderungen geeignet sind.

Acknowledgments

First of all, I thank my supervisor Prof. Dr. Philipp Slusallek and Prof. Dr. Hans-Peter Seidel for providing an outstanding research environment, for encuragement and support.

Then I like to thank my colleagues Ingo Wald, Heiko Friedrich, Stefan Popov, and Tongbo Chen. It was a pleasure researching and working with you.

Finally, I want to thank my wife for backing all the time. I love you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem and Motivation

Ray tracing has long been known to be the method of choice to produce convincing and realistically looking computer generated images of virtual worlds. Likewise, it has been known for its long rendering times, taking hours for just a single image.

Fortunately, rendering times improved significantly in recent years. With the growing computational power and fine-tuned algorithms, ray tracing became fast enough for interactive rendering. Even non-trivial scenes with millions of triangles can be ray traced at real-time frame rates for full-screen images. Nevertheless, most fast ray tracing techniques only support very simple shading and in particular only static walkthrough applications.

However, this new ability of real-time ray tracing rouses the desire for more interaction: to not only change camera parameters, but to also *change the geometry* of the scene between frames. Additionally, there are many scientific or entertaining applications – such as games or visualization of fluid simulations – that require dynamic scenes and interactive changes to the geometry. These applications have previously been the domain of the hardware-assisted rasterization algorithm but could tremendously profit from the ease of use and from the photorealism of ray tracing – if ray tracing would support dynamic scenes.

Unfortunately, changing the scene contents between frames is problematic, because fast ray tracing relies on preprocessing the geometry into a spatial index structure. The speed of this preprocessing phase was traditionally not so important, because rendering times were much higher than the preprocessing times. Therefore dynamically changing scenes were not supported by ray tracing systems nor investigated by researches in the past.

However, the situation changed: dynamic scenes are now interesting and within reach. Games could not only use advantages of ray tracing for visual realism as sketched above, but also interact with a *fully* dynamic world, freely changing the en-

tire environment. Or ray tracing could interactively visualize the results of physics simulations such as elastic deformations, cloth simulations, explosions, or smoke and fluid simulations. Consequently, several researchers around the world are attracted to the *problem of ray tracing dynamic scenes* – including ourselves. Thus, the work presented in this thesis deals with several approaches to handle dynamic scenes.

1.2 Contributions

Parts of this thesis have already been published elsewhere [Günther06a, Günther06b, Popov06, Günther07, Wald09].

The main contributions can be summarized as follows:

- We present one of the first methods for ray tracing non-rigidly deforming scenes interactively. The *motion decomposition* approach supports predefined animations as well as skinned meshes, and can be adapted to other kinds of deformations.
- We present a novel approach to quickly construct high-quality acceleration structures. With *streamed binning* we can build acceleration structures one order of magnitude faster than with previous methods.

1.3 Outline of Thesis

After establishing a background of ray tracing and its related problems in Chapter 2 we present two approaches to ray tracing dynamic scenes. Chapter 3 introduces motion decomposition as first approach. The second approach in Chapter 4 deals with faster construction of acceleration structures by using streamed binning.

Chapter 2

Background

In this chapter we will give a brief overview of ray tracing in general and its evolution and trends. We will discuss several possible approaches to ray tracing dynamic scenes to establish a context for our work. Additionally, we will also review alternative methods in more detail.

2.1 Ray Tracing

The ray tracing algorithm simulates optical transport of light to generate images of virtual worlds. But ray tracing is also the more fundamental function that simply returns the closest intersection of the scene geometry with a ray for a given ray origin and ray direction. It is important to distinguish between these different meanings of the term ray tracing: between the more general method of (recursive) ray tracing [Whitted80, Cook84] – which tracks for each pixel the path of light from the camera into the scene to the light sources – and the ray tracing function, that is used for these tracking tasks. In this thesis we mainly focus on the ray tracing function, and in particular on the construction of the acceleration structures that are necessary for its efficiency.

We assume a basic understanding of the ray tracing algorithm. For a deeper introduction and overview of ray tracing see e.g. [Shirley03, Pharr10].

2.1.1 Acceleration Structures

One of the main components of an efficient ray tracing function is the acceleration structure. The acceleration structure spatially indexes the primitives of a scenes and can then be queried to greatly reduce the number of ray-primitive intersection calculations. Many different data structures for accelerating ray tracing have been proposed including, among others, octrees [Glassner84, Arvo88], bounding volume hierarchies

(BVHs) [Rubin80], grids [Cleary83, Amanatides87], ray classification [Arvo87], binary space partition (BSP) trees [Fuchs80, Ize08], and kd-trees [Jansen86, MacDonald90, Subramanian90]. With the renewed interest in ray tracing several hybrid data-structures and variants have been proposed, such as the s-kd-tree / bounding interval hierarchy (BIH) [Ooi87, Wächter06, Havran06], the b-kd-tree [Woop06], the H-tree [Havran06], the RBSP-tree [Kammaje07], and BVHs with a higher branching factor then two [Dammertz08a, Ernst08, Wald08a].

All these acceleration structures have different properties, advantages and disadvantages. They differ in construction time and complexity, query time, implementation complexity, their ability to adapt to the scene geometry, and memory requirements, among others.

Based on a detailed evaluation of acceleration structures Havran concluded that in general kd-trees perform best or are at least highly competitive [Havran01]. As a result, many real-time ray tracing systems use kd-trees, including CPU-based systems [Wald01, Reshetov05, Overbeck07, Georgiev08, Djeu11], GPU-based ray tracers [Foley05], and ray tracing hardware [Woop05].

However, Havran's evaluation concentrated on static scenes and thus excluded construction time. For dynamic scenes the situation may change completely. For example, it has been shown that BVHs are easier to update after geometry changes [Lauterbach06, Wald07b, Yoon07]. BVHs can be quickly constructed as well, as we will discuss in Chapter 4. Thus BVHs seem to be the better suited acceleration structure for animated scenes. Regarding ray tracing performance it has been demonstrated that BVHs built according to the surface area heuristic (SAH) [MacDonald90] are quite competitive to kd-trees, in particular if groups of rays are traversed together [Wald07b, Benthin12]. Furthermore, BVHs are successfully used for GPU ray tracing [Aila09, Parker10, Garanzha10] and mobile ray tracing hardware [Nah11, Lee13].

2.2 Types of Motion

Firstly we want to distinguish between different *kinds of motion* that can be encountered in a dynamic scene before we look into their implications on acceleration structures.

First, there are *static* objects which are neither moved nor deformed. Static objects can be rendered very quickly with ray tracing using kd-trees [Reshetov05, Wald01, Havran01].

Second, there are objects which undergo simple affine transformations like translation or rotation. Lext and Akenine-Möller [Lext01] proposed a hierarchy of oriented bounding boxes to minimize reconstruction times via lazy rebuilds for this kind of *rigid-body motion*. Furthermore, Wald et al. [Wald03] have proposed a two-level kd-

tree scheme based on hierarchically affine transforming sub-meshes. Though this is sufficient for several engineering-style VR applications, more general animations cannot be handled.

Third, there is *locally coherent deformation*. This type of motion can be plentifully encountered in the real world and is thus widely simulated in graphical systems such as games and animated movies. For example, locally coherent deformation can be observed when wind bends a tree and the leaves are rippling, or when fingers are moved and their skin stretches.

Finally, our last category is completely *random motion* like in particle systems and turbulence simulations. This is the hardest type of motion to deal with, because in general no assumptions can be made anymore: the topology of the objects may change and one cannot rely on spatial or temporal coherence.

2.3 Approaches to Ray Tracing Dynamic Scenes

Research in the past focused on making the acceleration structure more efficient in culling unnecessary intersection test – and achieved outstanding results. For example, a well prepared kd-tree can reduce the number of necessary ray-triangle intersections to only two or three until the closest intersection is found [Havran01].

Ironically, these highly-efficient indexing methods are the source of the problem of ray tracing *dynamic scenes*. Dynamically changing scenes invalidate the pre-built acceleration structure and rebuilding it is expensive and takes several seconds, even for a moderate number of primitives, which is clearly not fast enough for interactive frame rates.

This leads us to several approaches to nevertheless ray trace dynamic scenes:

1. *Avoid the reconstruction* of the acceleration structure by ensuring that certain changes of the geometry of the scene do *not* invalidate the acceleration structure.

2. Develop *faster construction* algorithms that reduce the time to build the acceleration structure, such that it can ultimately be built several times per second to adapt to scene changes.

3. *Refit or update* an acceleration structure instead of rebuilding it from scratch when the scene changes. BVHs can be easily and quickly adapted to changing geometry by just recomputing the bounds of all nodes. However, because the hierarchy itself is not changed, the quality of the BVH quickly deteriorates and ray tracing becomes slower and slower after several of such updates. Such deterioration can be detected and, if above a threshold, the BVH is rebuilt from

scratch [Lauterbach06]. However, the reconstruction of the BVH leads to a disruptive pause in rendering, which can be avoided when the reconstruction is performed asynchronously to refitting and rendering [Ize07, Wald08b].

Another strategy to mitigate the quality degradation of BVH refitting requires knowledge of the changes to the scene to some degree beforehand. Then, the BVH can be constructed over a range of an animation or over different poses, such that the structure of the BVH is somewhat adapted to all these changes and not only to one instance of the scene [Wald07b].

Although refitting alone is often not enough to retain BVH quality, completely rebuilding the tree is also often not necessary. Instead, it can be sufficient to locally update the BVH structure to react to changes of the scene [Yoon07, Eisemann07, Kopta12]. The fast BVH update scheme of Bittner et al. [Bittner13] could be applied to animated scenes as well.

4. *Improve the culling performance* of acceleration structures that can be generated very quickly, but have previously been abandoned because of poor ray tracing performance. Probably the prime example of this approach is the grid. Constructing a grid is relatively simple and can be done in $\mathcal{O}(n)$. Very fast construction is possible by using multiple CPUs [Ize06] or the GPU [Kalojanov09]. Already simple to traverse with single rays, traversal of grids can be significantly accelerated by exploiting ray coherence [Wald06b]. However, grid performance suffers a lot if the geometry is distributed non-uniformly – as is the case in most real world scenes. A common solution is to nest grids, which also has been demonstrated recently for GPU ray tracing [Kalojanov11].

   To some degree the approach to improve ray tracing performance for accelerations structures that are better suited for dynamic scenes also applies to BVHs. Scenes that have both small and large triangles are problematic for BVHs, because their nodes need to be at least as large as the largest primitive they bound. This can lead to considerably overlapping bounding boxes and consequently to poor ray tracing efficiency. Kd-trees do not suffer from this problem, because they partition space instead of objects and simply split large primitives. BVH performance has been improved by adopting this property of kd-trees, by either splitting large triangles before BVH construction [Ernst07, Dammertz08b], or by introducing additional spatial splits during the construction of BVHs [Stich09, Popov09].

5. Develop *hybrid* acceleration structures that try to combine good construction performance and good culling performance. The s-kd-tree and its variants [Ooi87,

Wächter06, Woop06, Havran06] are such hybrid acceleration structures. By having two (or four) parallel planes per node to bound the children in one dimension they combine properties of the kd-tree and the BVH.

6. Use *no explicit acceleration structure* at all. Recently, a new approach to ray trace dynamic scenes has been introduced, called divide-and-conquer ray tracing [Mora11, Áfra12]. All rays to be intersected are gathered first and are processed together. The bounding box of the scene is recursively split, and each split partitions the rays as well as the primitives of the scene. Divide-and-conquer ray tracing adapts nicely to the distribution of the rays – parts of the scene that are not visited by rays are not further subdivided – and works also well for incoherent rays.

In this thesis, we focus on the first two approaches, *avoid the reconstruction* and *faster construction*, which will be discussed in detail in Chapter 3 and Chapter 4, respectively.

Chapter 3

Motion Decomposition

In this chapter we present our motion decomposition approach to ray trace dynamic scenes where deformations are locally coherent. After an overview of the motion decomposition framework we show in detail how to apply this general technique to predefined animations and to skinned meshes. At the end of this chapter we evaluate and discuss our proposed algorithms.

3.1 Introduction and Related Work

To approach the problem of ray tracing dynamic scenes we developed the motion decomposition framework with two goals in mind: On the one hand the proven kd-trees should be utilized as acceleration structure to allow for a high level of ray tracing performance. On the other hand updating or costly reconstruction of kd-trees in order to adapt to changing geometry should be avoided (while still ensuring correct ray tracing).

The key for achieving these goals can actually be found in the notion of relativity of motion: whether one moves an object filmed by a camera to the left or one moves that camera to the right will not change the result on screen, which is showing the object moving left. This principle equally works with rays that intersect moving objects: the intersection result will be the same regardless of whether the object is moved or the ray moved in the opposite direction.

Lext and Akenine-Möller [Lext01] have been the first to apply this principle to ray tracing animated objects. Every potentially moving object has its own acceleration structure (a recursive grid), and all objects (with their current position and orientation) are organized in an oriented bounding box hierarchy. The same idea was later adapted by Wald et al. [Wald03] to solely use kd-trees. Although ray tracing of animated scenes was realized, both implementations can only deal with rigid-body animations.

With the motion decomposition approach we further generalize this basic principle to non-rigid-body transformations. To this end we need to exploit knowledge about

the motion in the scene, and we need to make certain assumptions about the scene properties.

First, we assume that objects are defined as deformations of a base mesh. This means that the connectivity of the mesh remains the same during dynamic changes and only the vertices are animated. Second, we assume that all possible deformations of a mesh are bounded and that this bound can be determined in advance. Third, we assume that the motion is locally coherent, i.e., vertices that are topologically close to each other should have similar trajectories.

These assumptions hold for a variety of real-world animations. Vertex animations are quite common in visualization and games. If the animation is predefined our second assumption is automatically fulfilled. But we also show that interactive deformation which are defined through skinning of a base skeleton – as typically done for game characters – are supported by our method.

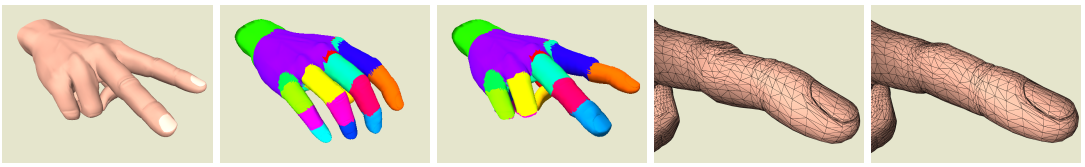
3.2 Method Overview

The concept and motivation of the motion decomposition framework is sketched in Figure 3.1. We exploit locally coherent motion of an animation (a) by automatically clustering coherently moving parts of the scene together in a preprocessing step (b). The common motion of these parts can be expressed by affine transformations. Using only these affine transformations to transform rays during ray tracing can quite closely reconstruct the original animation (c). However, the close-up (d) reveals cracks and other errors, because affine transformations alone cannot represent non-rigid body motion. Therefore we also account for the residual motion, which is handled by so called *fuzzy kd-trees*. Decomposing the motion into affine transformations and residual motion yields an correctly animated mesh (e).

In the following we describe in more detail the individual parts of our approach.

3.2.1 Decomposition of Motion

In Figure 3.2 we exercise the decomposition of motion for the example animation of a ball that is thrown onto a floor. The motion decomposition approach requires that the motion is locally coherent. This is the case for the floor (which is coherently "not moving") as well as for the ball. Therefore we can decompose the motion of these two objects into two parts: *affine transformation* and *residual motion*. For the floor these two parts are trivially zero. "Subtracting" the affine transformation for the ball yields a local coordinate system in which the (residual) motion of the vertices is much smaller – it represents now only the deformation without the higher-level movement of the ball.



(a) (b) (c) (d) (e)

Figure 3.1: *Motion decomposition together with fuzzy kd-trees allow for ray tracing deforming meshes by decomposing the motion of the mesh into an affine transformation plus some residual motion. (a) One frame of an animated hand. (b) The deforming mesh is split into sub-meshes of similar motion, shown in the rest pose. (c) Reconstruction of frame (a) using the affine transformations of each cluster only. (d) Close-up view of (c) revealing the erroneous mesh when approximated only by affine transformations. (e) Adding the residual motion handled by the fuzzy kd-trees yields the original mesh.*

Note that affine transformations can also compensate the shearing of the ball in the third frame of the animation (green), which yields smaller bounding boxes in the local coordinate system.

3.2.2 Fuzzy KD-Trees

The residual motion is only in rare cases completely zero (e.g. when there are only rigid-body transformations). Thus to generally handle residual motion we introduce *fuzzy kd-trees*. First, the residual motion of each triangle is bounded by a *fuzzy box*, a box in the local coordinate system that encloses the motion of the three vertices of the triangle. Then a kd-tree is built over these fuzzy boxes instead of just the triangles, resulting in a fuzzy kd-tree. As long as we assure that the fuzzy boxes are not violated by too strong residual motion the fuzzy kd-trees stay valid even during mesh animations (see Figure 3.3). Thus fuzzy kd-trees can be built in a preprocessing phase and do not require rebuilding during animation. This is the key benefit of our approach.

3.2.3 Two-level KD-Trees

Because the relationship between the coherently moving parts of an animation and their motion is determined by affine-only transformations we use a two-level acceleration structure in the spirit of [Lext01] and similar to [Wald03]. For each frame to be rendered we update the transformations and current bounding boxes of objects having an own fuzzy kd-tree (shown in Figure 3.4). Then a small top-level kd-tree is built over these bounding boxes. Only this top-level kd-tree needs to be reconstructed ev-

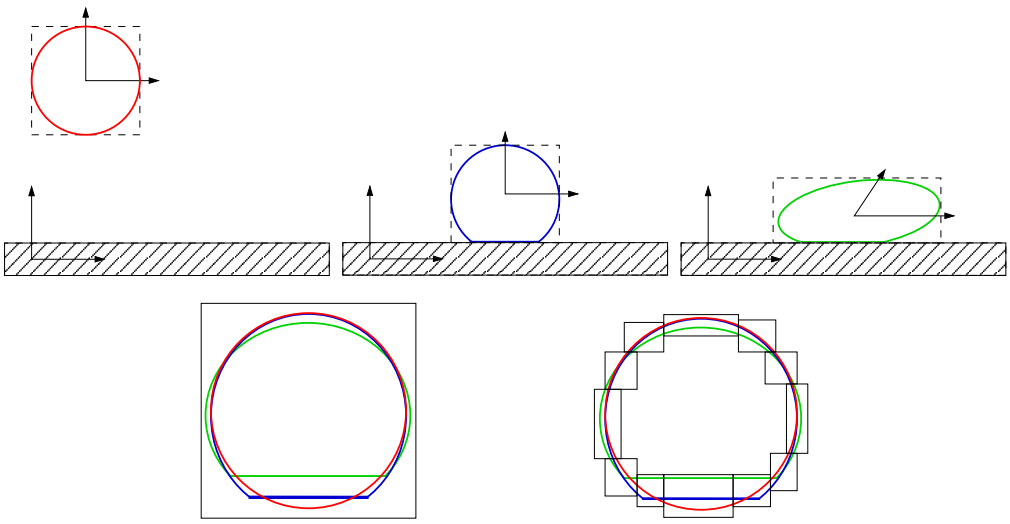


Figure 3.2: *Example of a motion decomposition. Top row: Three frames of an animation where a ball is thrown onto a floor, together with the bounding boxes and local coordinate systems of the two objects. The motion of these objects is encoded by affine transformations. Bottom row: Visualization of the bounded residual motion in the local coordinate system of the ball – coherent dynamic geometry is now "almost static".*

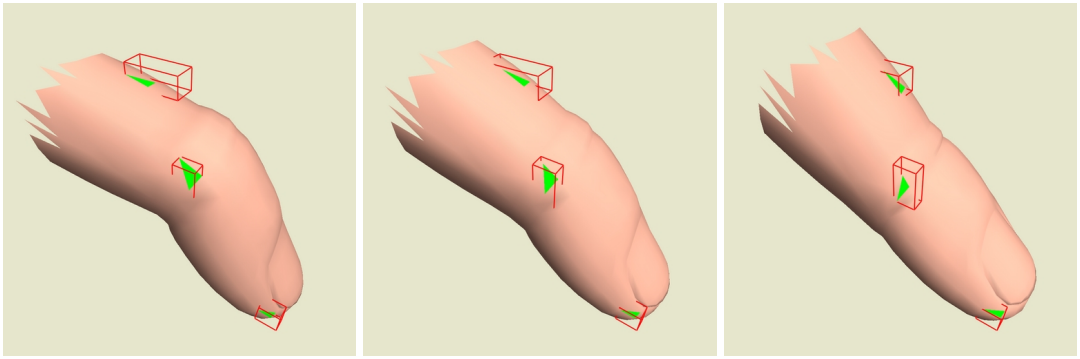


Figure 3.3: *The triangles of the index finger of the* HAND *animation are clustered together and are shown here. The residual motion of each triangle (green) is bounded by a fuzzy box (red). Although the triangles move a little bit in the local coordinate system their fuzzy boxes do not change. As the fuzzy kd-tree is built over these fuzzy boxes instead of the triangles it is valid for* all *time steps.*

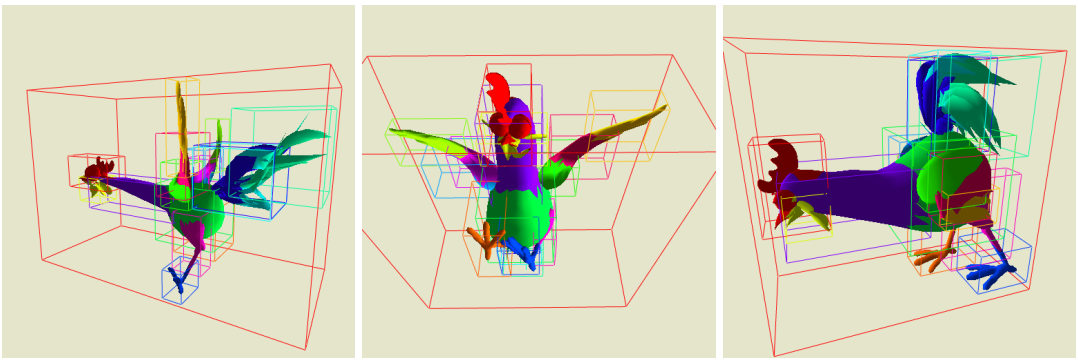


Figure 3.4: *To ray trace a frame of an animation we firstly build a small kd-tree over the current bounding boxes of each cluster. These boxes are shown for three frames of the* CHICKEN *animation. Rays hitting a cluster get inversely transformed into the local coordinate system of this cluster and traverse its associated fuzzy kd-tree.*

ery frame, which can be done very quickly because the number of moving objects is usually small (say 100 or 1000).

Ray traversal of this two-level kd-tree scheme works as in [Wald03]. When ray traversal of the top-level kd-tree reaches a top-level leaf the rays are transformed into the local coordinate system of the contained objects. Subsequently they continue the traversal of the corresponding fuzzy kd-tree.

When rays finally visit a leaf of a fuzzy kd-tree they must be tested against the *current* instantiation of any contained triangle. These triangle intersection tests can be done in world coordinate system with the untransformed rays. Thus the positions of the triangles in the local coordinate system are not explicitly required for rendering, which saves memory and computation time.

3.3 Motion Decomposition of Predefined Animations

The general approach of motion decomposition holds for predefined animations and skinned meshes. However, the way how coherently moving triangles are found and clustered together is quite different for both scenarios. Therefore we explain the two specialized clustering algorithms separately.

3.3.1 Details of Motion Decomposition

Before we describe the clustering algorithm we designed for predefined animations we first need to specify its input and goals more formally. This section discusses the details

of the motion decomposition approach and fuzzy kd-trees and provides a more formal description of the techniques involved.

Our method requires that the animation is defined as deformations of a base mesh, i.e. that the connectivity of the mesh is the same at any time. Thus the animation is given by a constant set of triangles $\{\Delta\}$ and their vertex positions $\{v\}$. The trajectory $v_i(t) \in \mathbb{R}^3$ describes the motion of a vertex over time. No additional knowledge about the deforming mesh is necessary. However, we inherently assume coherent local motion, i.e., vertices that are topologically close[1] to each other should have similar trajectories.

We exploit this coherent motion of the mesh to "subtract" common motion because the resulting smaller residual motion can be better handled by fuzzy kd-trees, thus improving performance of ray tracing the animation. Mathematically this motion decomposition can be described as follows. The position v at time t of the vertices in world space can be expressed by applying the appropriate affine transformation X to a *rest pose* $\tilde{M} = \{\tilde{v}\}$ plus a (world-space) residual motion δ:

$$v_i(t) = X(t) \cdot \tilde{v}_i + \delta_i(t) \tag{3.1}$$

The rest pose of an animated object is usually the state of the mesh during modeling before a skeleton or an animation is applied and where typically all parts of the object are in a relaxed and exposed position.

Transforming into the coordinate system of the rest pose by multiplying (3.1) with the inverse transformation $X^{-1}(t)$ and substituting $\tilde{\delta}_i(t) = X^{-1}(t) \cdot \delta_i(t)$ yields

$$\tilde{v}_i(t) = \tilde{v}_i + \tilde{\delta}_i(t) = X^{-1}(t) \cdot v_i(t), \tag{3.2}$$

the *fuzzy position* $\tilde{v}_i(t)$ of vertex i at time t. The *fuzzy box* $\mathrm{FB}(\Delta_{abc})$ of triangle Δ_{abc} with vertices a,b,c is the axis-aligned bounding box (AABB) of all fuzzy positions $\tilde{v}_a(t), \tilde{v}_b(t), \tilde{v}_c(t) \forall t$ and bounds the local residual motion $\tilde{\delta}(t)$. The fuzzy kd-tree is then built over these fuzzy boxes instead of the triangles and is thus valid for *all* time steps.

3.3.2 Clustering for Predefined Animations

The performance of the fuzzy kd-tree for ray tracing is strongly dependent on the size of the fuzzy boxes, because the surface area of the fuzzy boxes is proportional to the probability that a random ray will hit this box [MacDonald90]. In other words, if the fuzzy boxes are large – due to high "fuzziness" of the triangles they bound – they

[1] There are short paths in the mesh that connect these vertices.

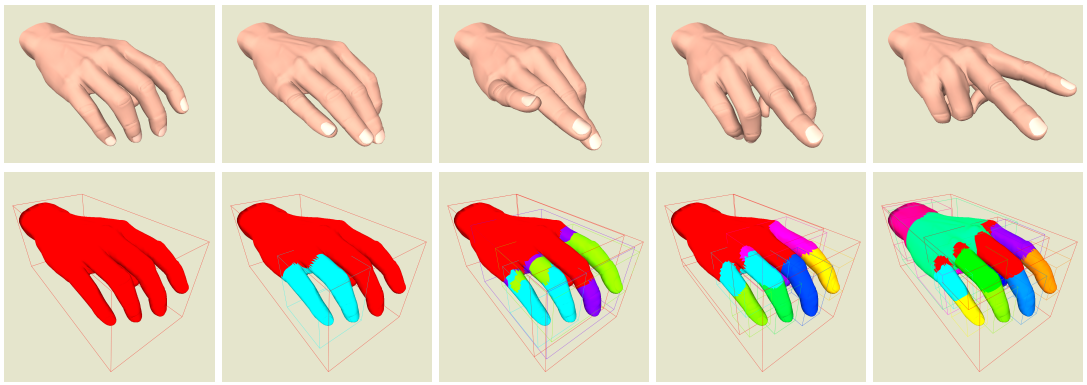


Figure 3.5: *Top row: Several frames of the* HAND *animation. Bottom row: Clustering of the* HAND *animation with 1, 2, 4, 8, and 12 clusters, respectively. Triangles with the same color belong to the same cluster. The wire frame boxes denote the bounding box of each cluster. Note how the cost-driven clustering approach automatically segments the hand into clusters in which the triangles perform roughly the same motion. For example the pinky and the ring finger are clustered first as they perform similar motion.*

will overlap significantly. Such overlap is the reason that the acceleration structure can avoid less intersection computations, resulting in many costly ray-triangle intersections until the closest intersection is found.

Therefore we like to minimize the residual motion, or equivalently, we want to subtract as much common motion as possible. This can only be achieved by exploiting the *local* coherence in motion, which requires a clustering of the mesh into sub-meshes that deform coherently.

To partition the triangles of the mesh into clusters we apply a generalized Lloyd relaxation [Lloyd82] algorithm (see e.g. [Du99] or [Gersho91] for an introduction). Note that we need to cluster triangles and not vertices because the triangles are the primitives which will be tested for intersections with rays.

In each iteration step we firstly find affine transformations that minimize the residual motion of each cluster and subsequently reassign each triangle to the cluster where its residual motion is smallest.

The iteration process stops when the clustering converged, i.e. when no triangle changes its cluster anymore. Practically we stop when the decrease in the overall residual motion drops below some threshold, e.g. below 1‰.

As Lloyd relaxation is prone to find local minima and as the optimal number of clusters is not known in advance we start with one cluster and iteratively insert a new cluster until the cost function converges.

A new cluster is inserted in the following way. We take as seed triangle the triangle

with the largest residual motion. We also include its neighboring triangles such that they define a unique coordinate system. The existing clusters are also newly initialized with the seed triangles that have the *smallest* residual motion in each cluster. These seed triangles act as prototypes of the common motion of the (currently) clustered triangles and ensure a stable clustering procedure. We stop adding clusters when overall residual motion cannot be reduced significantly anymore.

The clustering process is demonstrated in Figure 3.5. Note that the most similar moving parts of the mesh are clustered first. The proposed clustering algorithm is summarized in pseudo code in Algorithm 1.

Algorithm 1 Clustering

```
start with one cluster containing all triangles of the scene
while global cost still changes significantly do
    while cost not converged for current #clusters do
        find transformations, minimizing δ̃ ∀cluster
        recluster triangles, minimizing δ̃
    end while
    create new cluster
    choose seed Δs
end while
```

3.3.3 Finding "Good" Transformations

For the motion decomposition and the clustering we need to find affine transformations between the rest pose and all other poses that minimize the residual motion $\|\tilde{\delta}_i(t)\| = \|X_t^{-1} \cdot v_i(t) - \tilde{v}_i\| \; \forall t$.

The affine-only 3×3 transformation matrix and a translation can be combined to a 4×4 transformation matrix using the homogeneous coordinate system. Thus X^{-1} has 12 unknown elements and we need (at least) four linearly independent vertex positions for a unique solution. With more than four vertex positions (the general case) we solve the linear least squares problem that minimizes the L^2-norm $\|X^{-1} \cdot v_i(t) - \tilde{v}_i\|_2$ of the residual motion $\tilde{\delta}$.

Instead of minimizing the L^2-norm of the residual motion we actually want to optimize the surface area of the resulting fuzzy boxes. This is more closely linked to the L^∞-norm. But we found that using the L^2-norm is already giving good results while guaranteeing convergence.

3.3.4 Selecting the Rest Pose

So far we have not yet defined the rest pose $\tilde{M} = \{\tilde{v}\}$. To avoid an expensive general optimization problem we simply take as $\tilde{M}$ the positions $\{\tilde{v}_i(t)\}$ of one of the key frames of the animation. During the clustering we try all key frames and take the one that minimizes the sum over all cluster c and all time steps t:

$$\tilde{t} = \arg\min_{t'} \sum_t \sum_c \sum_{i \in c} \| X_c^{-1}(t) \cdot v_i(t) - v_i(t') \| \tag{3.3}$$

If the animation consists of a large number of frames we uniformly subsample the set of frames while searching for an optimal rest pose in order to reduce the search space.

3.3.5 Optimizing Local Bounding Boxes

We can make two observations that can lead to better ray tracing performance if exploited properly.

First, when rays hit the axis-aligned bounding boxes of the clusters in the top-level kd-tree they get inversely transformed to the local coordinate system of that cluster. The following traversal of its fuzzy kd-tree starts by clipping the transformed rays to the *local* AABB of the cluster. This is meaningful because the affine transformation can rotate the coordinate systems and thus rays hitting the global AABB do necessarily also hit the local AABB.

Second, the vertices of each cluster can be rotated arbitrary in the local coordinate system because this rotation can be compensated by multiplying the affine transformations of the corresponding cluster with a rotation matrix.

We can take advantage of these facts to make tighter local axis-aligned bounding boxes of clusters. By calculating an oriented bounding box (OBB) of the fuzzy positions and rotating this box to make it axis-aligned we can directly minimize the local bounding boxes.

To approximately compute the OBB of each cluster (in local coordinates) we apply principal component analysis (PCA) to the fuzzy positions and take the first three PCA vectors as new coordinate system axes. Exactly finding the OBB [Barequet01] could also be used.

3.4 Motion Decomposition of Skinned Meshes

Skinned meshes are a powerful tool for content creation in computer games and animated movies. Artists create meshes and attach a bone model to it. Using this skeleton

(and optionally a physical behavior model) they can apply deformations to the mesh. This type of motion is a good approximation of many real scenarios and is also common for animated synthetic datasets. Skinned meshes provide many advantages over predefined animations, such as more flexibility and artistic freedom as well as being more memory efficient. Thus our goal is to also ray trace skinned animations – with the introduced motion decomposition approach. As we will see, the motion decomposition and clustering task is simpler for skinned animations: The affine transformations are directly provided from the application in form of bone transformations.

3.4.1 Vertex Skinning

A skinned mesh is animated with the help of an underlying skeleton, usually referred to as skeleton subspace deformation [Magnenat-Thalmann88, Magnenat-Thalmann91]. Each vertex of the mesh is influenced by one or more bones. For example, vertices forming an arm move together with the corresponding arm bone. Additionally, vertices near the joints of two bones are influenced by both bones, resulting in a smooth skin surface without cracks or other artifacts near the joints.

Mathematically, bones are described by transformations, and standard skinning is a simple weighted interpolation:

$$v = \sum_{i=1}^{n} w_i X_i \tilde{v}, \quad \text{with} \sum_{i=1}^{n} w_i = 1$$

The value n is the number of bones influencing the skinned position v. X_i is the current transformation of bone i and w_i is the associated scalar weight (or influence) of that bone for the vertex $\tilde{v}$ in the rest pose of the mesh. The rest pose is the original mesh configuration modeled by an artist without applied skinning.

If all vertices were to be influenced by only one bone ($n = 1$) we would have the situation of simple affine-only transformations where the two-level kd-tree alone is sufficient [Wald03].

However, it is an important observation that even when $n > 1$ the interpolated motion of a vertex in the *local coordinate system* of a carefully chosen bone is typically rather small, because usually vertices are significantly influenced by neighboring bones only. This small residual motion of the vertices and triangles can be bounded by the fuzzy boxes. Then for each bone, a fuzzy kd-tree is built over the fuzzy boxes rather than over the triangles. Again, this has to be done only once in a preprocessing step, because the local movement of the triangles does not invalidate the fuzzy kd-tree (Figure 3.3).

To render a new frame the pose of the skeleton is updated as well as the vertex positions. Then the bounding boxes of each bone and associated triangles are calculated

in world-space. Finally, we can apply the two-level kd-tree scheme and rebuild the small top-level kd-tree over the bone fuzzy kd-trees.

3.4.2 Stochastic Fuzzy Box Computation

For the correctness of this algorithm it is crucial that the fuzzy boxes used to build the fuzzy kd-trees are conservative. Otherwise it might happen that triangles get culled during traversal of the fuzzy kd-trees and that rays erroneously miss these triangles, which will result in holes in the mesh. Additionally, to optimize ray tracing performance, we prefer to have small fuzzy boxes.

To estimate a set of small, but sufficiently conservative fuzzy boxes, we sample the pose space during a preprocessing phase. The full pose space can be sampled by rotating each bone arbitrarily. However, tighter fuzzy boxes and thus better ray tracing performance can be achieved by restricting the bone rotation relative to its parent bone. Additional information from the rendering application – such as joint limits – can be used to restrict the pose space, because arbitrary bone rotations are quite unnatural for most models. Another possibility is to only sample the different animation sequences – such as walking, running, or kicking – used e.g. by a game application.

Note that the size of the fuzzy boxes depends not only on the pose space but also on the choice of the bone a triangle is assigned to. In order to find the optimal bone, we do an exhaustive search by calculating the fuzzy box in every bone's local coordinate system. The fuzzy box of a triangle is the union of the fuzzy boxes of its vertices.

The fuzzy boxes of all vertices are calculated by sampling as follows. For each sample (i.e. a pose of the skeleton), every vertex is skinned and subsequently transformed into the local coordinate space of each bone. The fuzzy box of a vertex in a bone's local coordinate system is the axis-aligned bounding box of its samples, i.e. its transformed positions.

3.5 Results

We implemented the motion decomposition approach and the ray tracer with fuzzy kd-trees using C++ and SSE [Intel02]. The kd-tree traversal is loosely based on *multi-level ray tracing* and the *inverse frustum culling* approach of Reshetov et al. [Reshetov05] with a ray packet size of 4×4. The ray-triangle intersection test is basically the same as described in [Wald04] for packets of rays.

We used the LAPACK library [Anderson99] for fast matrix operations and linear algebra computations needed by the clustering step for predefined animations. To manage the skeleton and vertex skinning parameters of the models we used the open source

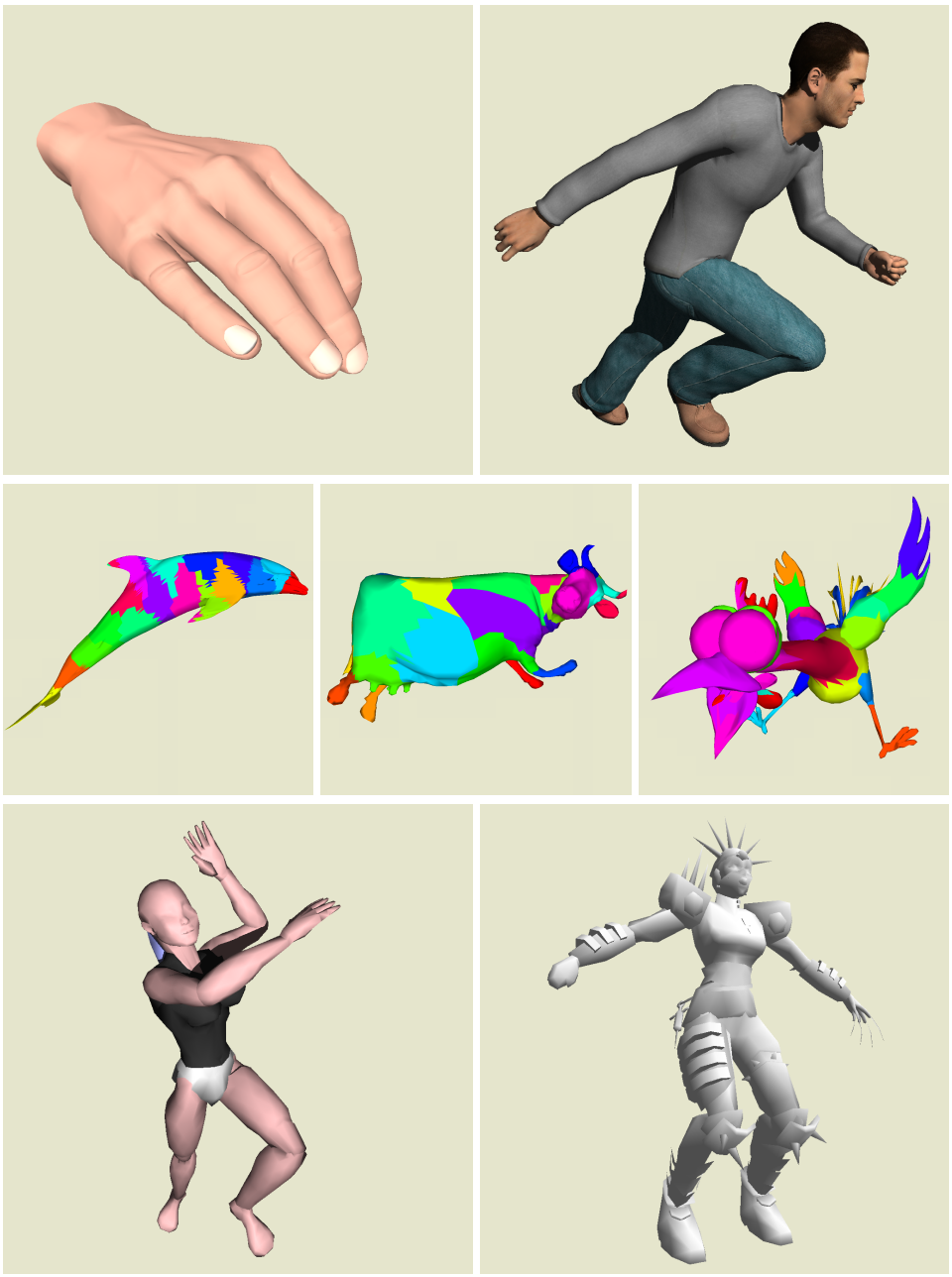


Figure 3.6: *Example animations for motion decomposition. The first two rows show predefined animations:* HAND *(16855 triangles, 30 frames),* BEN *(78029 triangles, 30 frames),* DOLPHIN *(12377 triangles, 101 frames),* COW *(5804 triangles, 204 frames), and* CHICKEN *(5664 triangles, 400 frames). The last row shows skinned meshes:* CALLY *(3342–271k triangles) and UT2003 (2478–201k triangles).*

library Cal3D [Cal3D]. With Cal3D we can also smoothly blend different animation sequences as well as interpolate between key frames (see Figure 3.11).

The measurements were done on two dual-core AMD Opteron workstations, one with 2.4 GHz processors and one with 2.8 GHz processors. Each workstation was equipped with a GeForce 6800 GT PCIe graphics board.

Figure 3.6 shows several of our example scenes. They cover different amounts of animation complexity and contain between approximately 2k and 271k triangles. The HAND and BEN scenes are examples of hierarchical animations and were created using a skeleton with vertex skinning. However, this skinning information was not available after exporting the animations, thus we still treat them as predefined animations. Because the hierarchical motion of these scenes directly matches our assumptions we expect that our motion decomposition approach will perform well. Although the motion of the DOLPHIN is not controlled by a skeleton but by a mass-spring simulation it still moves naturally and thus also meets our assumptions.

The COW scene, on the contrary, shows highly unnatural motion. It is an example of an animation generated by a physics-based simulation of a cow consisting of very elastic material, resulting in "jelly-like", unnatural poses. As a stress test for our algorithm we include the CHICKEN animation. This section of an animated film is difficult because of the extreme, cartoon-like sequences around frame 250 (see Figure 3.6 and the middle of Figure 3.4) that include strong stretching of the chickens' neck and oozing of its eyes. Even though the motion in these two scenes violates the assumptions of our motion decomposition framework it still works surprisingly well.

Finally, the last two scenes consist of the skinned models CALLY and UT2003, as typically found in games. We were not able to acquire meshes with a high number of polygons that include the necessary skeleton and skinning parameters. To still be able to estimate the performance of our system for scenes with a high number of triangles we decided to additionally subdivide the low-resolution models several times.

3.5.1 Clustering of Predefined Animations

Our proposed clustering algorithm minimizes the residual motion by using the least square distances of the vertex positions in the local coordinate system as a cost measure. However, the *surface area* of the fuzzy boxes is the better measure for the expected ray tracing performance [MacDonald90]. Figure 3.7 compares these two measures during the clustering process. It shows that the least square distances by the relaxation-based clustering is highly correlated with the surface area of the fuzzy boxes, and both are minimized. Note that the absolute values of the cost functions cannot be compared as they are dependent on the density of the mesh and the scale of the scene. Furthermore, it can be seen that adding new clusters is not beneficial anymore at some point and

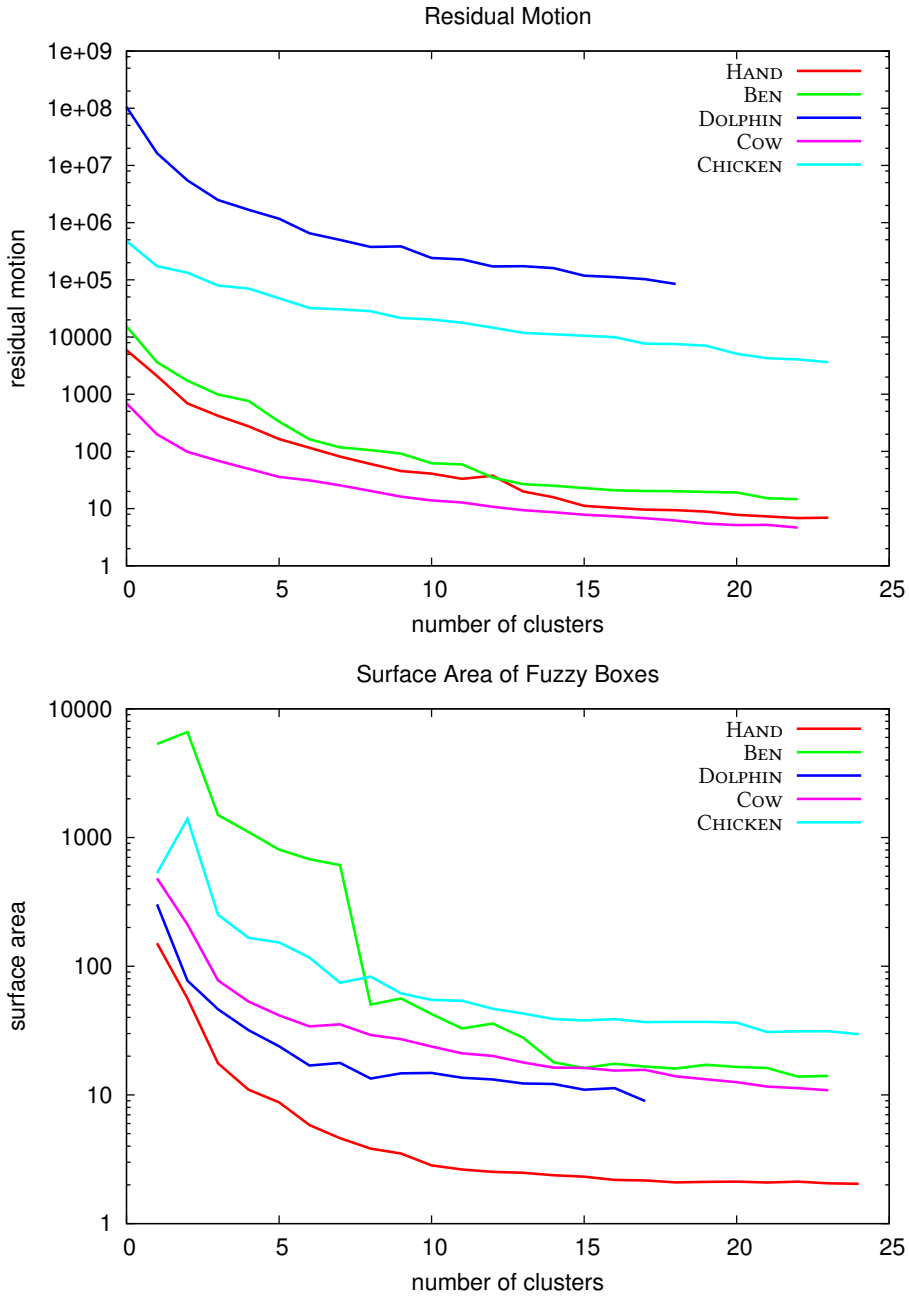


Figure 3.7: *The sum of all residual motion (top) and the accumulated surface area of the fuzzy boxes (bottom) during the clustering process. The residual motion is minimized by our clustering algorithm while the surface area of the fuzzy boxes is a better measure for ray tracing performance. Still, both measures correlate well.*

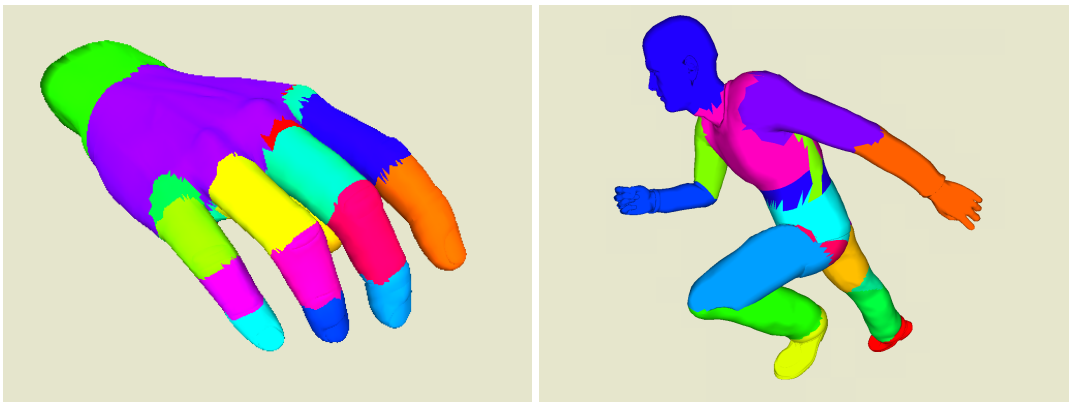


Figure 3.8: *Resulting clusters for the* HAND *(left) and the* BEN *(right) animation. The found clusters correspond well to the skeleton used to create the animations though it was not available as input to our algorithm.*

clustering is automatically terminated by a threshold on the change of the least square distances.

Also note that a number of clusters around 20–25 is sufficient for our example animations. The clustering process takes from about 20 minutes for the HAND to about 95 minutes for the CHICKEN when 25 poses were tested for the rest pose. The clustering time increases linear with the number of time steps and is also linear in the number of candidate rest poses. Thus it can be shortened drastically by sub-sampling the time steps during the clustering process with a negligible effect on clustering quality.

Although our algorithm only sees the animated vertex positions of the mesh and has no knowledge about the underlying skeleton, the resulting clusters correspond well to the bones of the HAND and BEN model (see Figure 3.8). Also for the DOLPHIN and even for the COW and CHICKEN animations meaningful sub-meshes that move coherently were found (see middle row of Figure 3.6).

3.5.2 Fuzzy Box Estimation for Skinned Meshes

To evaluate the trade-off between the flexibility of posing a skinned mesh and ray tracing performance, we restricted the valid pose space in several steps. Arbitrary rotations of the bones on the one hand allow the greatest freedom for a user to interact with the model. On the other hand, it is often meaningful to apply joint limits to the skeleton to avoid unnatural mesh deformations. The pose space can be further restricted by allowing only certain types of (predefined) movements (such as walking or waving), which is often the case in game applications.

pose space	#samples	fuzzy box surface area	avg. #tris per leaf	fps
arbitrary bone rotations	1000	747k	4.1	7.7
applying joint limits	245	580k	3.4	10.2
many animation sequences	124	552k	3.2	11.6
just one animation sequence	62	515k	3.1	12.0

Table 3.1: *Influence of the valid pose space on the fuzzy box surface area, the quality of the fuzzy kd-trees and subsequently the rendering performance for the UT2003 scene. Restricting the pose space – at least by joint limits – greatly improves ray tracing performance.*

Table 3.1 shows our measurements for all these pose space configurations. With increasing freedom of the bone rotations, the fuzzy boxes become larger. This leads to more overlap of the fuzzy boxes, which decreases the efficiency of the fuzzy kd-tree. More intersections need to be calculated, resulting in decreased ray tracing performance.

As already mentioned, the accumulated surface area of the fuzzy boxes of all triangles is a good measure for the expected ray tracing performance because it is proportional to the number of intersections of a random ray with these boxes [MacDonald90]. Another measure for the efficiency of kd-trees is the average number of triangles per kd-tree leaf. Higher numbers account for larger overlap of fuzzy boxes, because it is not meaningful to split the overlapping area during kd-tree creation – every child node would contain the same triangles – and we cannot save intersection tests. Finally, we verified our expectations by measuring the achieved frames per second for ray tracing the rest pose with the same viewpoint for all the different pose space configurations.

All these measures show that restricting the pose space improves the ray tracing performance. The biggest performance gain can be obtained by applying joint limits.

The time required for preprocessing and sampling the pose space can be found in Table 3.3. For the original skinned models it took less than a second on one 2.4 GHz Opteron CPU. And even with higher resolution meshes with hundreds of thousands of triangles we achieved reasonable fast preprocessing within seconds to a few minutes – which is two orders of magnitude faster than the clustering of predefined animations. Also note that sampling is usually faster than the subsequent building of the fuzzy kd-trees over the found fuzzy boxes.

	#traversal steps			#intersections			average fps		
scene	static	fuzzy	ratio	static	fuzzy	ratio	static	fuzzy	ratio
HAND	1,331k	2,308k	1.76	1.28	1.48	1.15	17.98	10.94	1.64
BEN	722k	1,112k	1.54	1.2	2.28	1.9	20.98	10.77	1.94
DOLPHIN	534k	1,020k	1.91	1.08	1.72	1.59	22.56	19.31	1.16
COW	634k	947k	1.49	0.92	3.68	4.0	19.21	12.49	1.53
CHICKEN	210k	379k	1.80	0.80	4.96	6.2	39.24	15.03	2.61

Table 3.2: *Comparison of the fuzzy kd-tree with a classic kd-tree in numbers of traversal steps and intersections (amortized per ray). We pay only a factor between 1.5 and 2 in the number of traversal steps and typically a factor of about 2 in the number of intersections for the ability to ray trace animations. Even the difficult CHICKEN and COW animations can be ray traced at interactive frame rates.*

3.5.3 Comparison to Static KD-Tree

To evaluate the efficiency of our proposed fuzzy kd-tree for ray tracing animated scenes we compare it to a static kd-tree.

For the predefined animations, Table 3.2 lists two characteristic values for acceleration structures, namely the number of traversal steps and the number of ray-triangle intersections. These values are given amortized per ray as we use frustum traversal on entire packets. We compare our two-level fuzzy kd-tree with an one-level kd-tree that is re-built and optimized for every time step of the animation. Apart from the structure of the resulting tree the traversal and intersection routines are the same for both kd-trees.

The measurements show that the number of traversal steps increases by only 50% to 100% for the fuzzy kd-tree. The ray-triangle intersections increase only by 15% for the HAND and typically up to 100%, except for the COW and CHICKEN scene. Because these two animations do not strictly obey our assumption of local coherent motion the motion decomposition cannot work optimally, resulting in larger residual motion and overlap of the fuzzy boxes, which eventually increases the number of intersections. Although the increase in intersections of a factor of 6 for the CHICKEN may seem high, overall ray tracing performance is much less affected (as shown in the following section) and ray tracing this animation is still reasonably fast.

Although not shown here, the situation is similar with skinned meshes: Ray tracing animations with fuzzy kd-trees takes about $2\times$ longer compared to using a static kd-tree for a single frame, which is a reasonable price for the added ability to handle flexible skinned models at interactive frame rates.

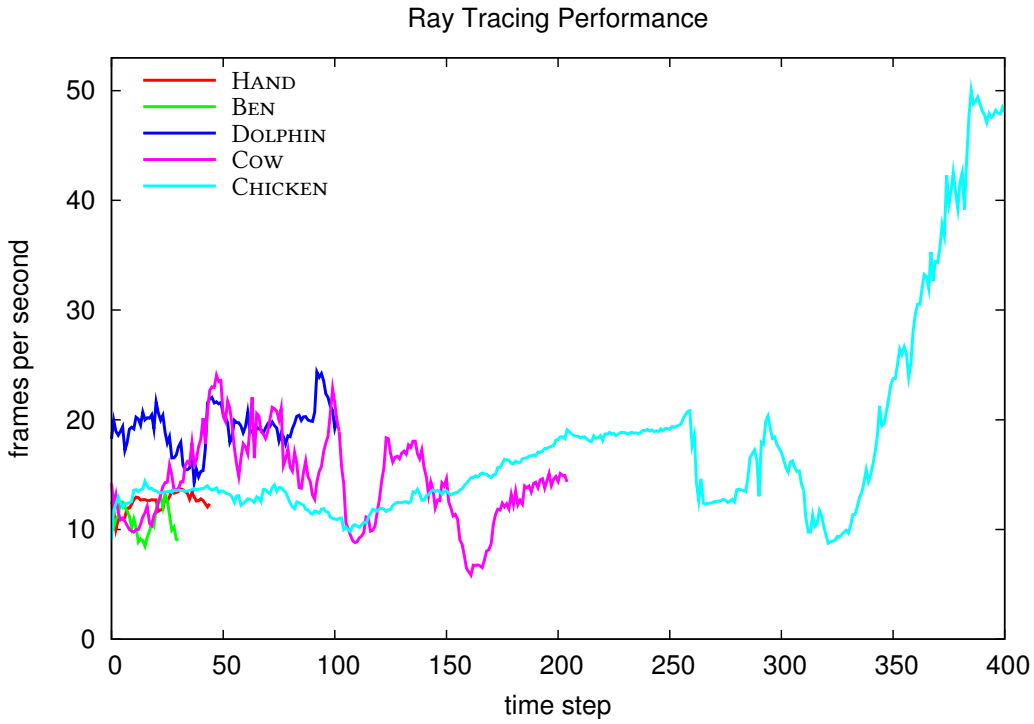


Figure 3.9: *Ray tracing performance for our predefined animations in frames per second on a single 2.8 GHz CPU including shading at 1024×1024 pixels. For all scenes at least interactive frames rates of 5 frames per second are achieved.*

3.5.4 Absolute Performance

In this section we present the overall performance of our ray tracing system for predefined animations and skinned meshes. All measurements were done at a resolution of 1024×1024 pixels with a simple diffuse shader and a packet size of 4×4 rays.

Figure 3.9 shows the frame rate achieved by our system over the course of each predefined animation. As can be seen we achieve interactive frame rates of 5 to 15 frames per second at full-screen resolution on a single 2.8 GHz CPU.

Additionally, Table 3.2 also compares the average frame rate achieved by ray tracing with fuzzy kd-trees to the performance of ray tracing with optimized static kd-trees pre-built for every frame. The overhead introduced by fuzzy kd-trees is between a factor of 1.2 and 2.6 and is roughly constant for a specific animation independent of the camera view or advanced shading using secondary rays (e.g. the shadows in Figure 3.10). Note that simply switching to a two-level kd-tree without any other changes already decreases ray tracing performance about 30% due to the costs of transforming

scene	#tris	preprocessing time in seconds sampling	building kd-trees	fps
CALLY	3k	0.63	0.65	9.5
CALLY	10k	1.80	2.28	7.1
CALLY	30k	5.21	8.77	5.7
CALLY	90k	16.4	34.5	3.6
CALLY	271k	50.4	150	2.0
UT2003	2k	0.59	0.39	15.5
UT2003	7k	1.23	1.67	12.3
UT2003	22k	3.08	6.41	7.4
UT2003	67k	8.63	23.8	4.0
UT2003	200k	29.7	88.9	1.9

Table 3.3: *Measured performance data of our skinned models for different subdivision levels. Even with hundreds of thousands of triangles we achieved reasonable fast preprocessing within a few minutes and still ray trace at interactive frame rates.*

the rays to the local coordinate system, restarted traversal, and the fact that a two-level kd-tree cannot adapt to the scene geometry as well as one single kd-tree.

Finally, Table 3.3 also reports our performance measurements for skinned meshes. When ray tracing these dynamic scenes on a 2.4 GHz CPU we achieved a rendering performance of up to 15 frames per second. Furthermore, interactive frame rates were still possible for large scenes.

3.6 Discussion and Future Work

The motion decomposition framework with the fuzzy kd-trees can efficiently handle dynamic scenes. As fuzzy kd-trees are essential kd-trees they can be used together with many kd-tree algorithms and share their outstanding performance properties – this was the main motivation of the motion decomposition approach and makes the integration of motion decomposition into an existing ray tracer easy. In Particular, besides traditional single ray traversal also packet [Wald01] or frustum traversal [Reshetov05] algorithms work with fuzzy kd-trees; layout and cache optimizations (e.g. [Yoon06]) can be applied to fuzzy kd-trees as well; advanced kd-tree construction algorithms can also be used for fuzzy kd-trees – including the algorithms we present in Chapter 4 – because they are constructed over fuzzy boxes, which are just ordinary boxes when

seen by a construction algorithm[2].

The use of a two-level acceleration structure (Section 3.2.3) and fuzzy kd-trees – as required by our approach – is about $2\times$ slower compared to use of a single kd-tree (which does not support dynamic changes). This overhead is caused by the reduced culling efficiency of the kd-trees due to overlapping object bounding boxes and/or overlapping fuzzy boxes, the time needed to rebuild the top-level tree, and the more complex ray traversal that includes the transformation of rays between coordinate systems.

However, using a top-level acceleration structure also provides additional advantages. For example, we can easily support instantiation of objects [Wald03]. Furthermore, the local coordinate system of the axis-aligned fuzzy kd-trees can be rotated to achieve better bounds and thus performance.

Larger scenes consisting of several independently animated objects should also be feasible. All clusters of all objects are organized into the top-level kd-tree. As long as the total number of clusters is sufficiently small (say < 10000) the rebuild of the top-level kd-tree will not become a bottleneck.

The motion decomposition approach is restricted to semi-hierarchical, locally coherent motion – handling random motion is not supported. Animations that violate our implicit assumption of locally coherent motion will still be ray traced correctly but with non-optimal performance.

The efficiency of our fuzzy kd-trees also depends on the level of tessellation. If the same animated mesh is tessellated to a higher number of triangles the overlap of the fuzzy boxes will increase. This leads to more ray-triangle intersections because the kd-tree cannot cull so much intersections as before. One promising idea to cope with this problem is to combine the fuzzy kd-trees with vertex culling [Reshetov07], which handles large fuzzy boxes with many triangles, resulting in less relative overlap of fuzzy boxes. Note that this approach would need a different cost function, as now not the overall residual motion should be minimized anymore, but ultimately the surface area of all fuzzy boxes.

Besides predefined animations we also support skinned meshes. Skinned meshes have several advantages over predefined animations:

- The preprocessing time is greatly reduced. For predefined animations an expensive clustering step is necessary to find coherent moving triangles.

[2]Note however, that optimizations like *perfect split* [Havran01] cannot be used, as these require triangles and not boxes as building primitive.

- Skinned meshes save a lot of memory because they just need one base mesh for the entire animation, plus the corresponding bones and skinning information – in contrast to many meshes, one for every time-step.
- The most important advantage is the gained flexibility: With skinned meshes it is no problem to smoothly interpolate between key frames or to blend several animation sequences. Even direct user interaction is possible if we sample the (optionally limited) pose space.

As an alternative to ray trace predefined animations with fuzzy kd-trees on could pre-build many static kd-trees, one for each frame of the animation. Obviously, the total data size using a separate static kd-tree for each time step will linearly increase with the number of frames, whereas the memory consumption of our fuzzy kd-tree will be roughly the same, largely independent of the length of the animation. Thus, motion decomposition can be seen as a kind of compression scheme, where the clustering algorithm extracts the redundant information. Additional memory could be saved by compressing the triangle information as well, either storing only the residual motion, or using lossy methods such as principal component analysis [Alexa00].

Another promising direction of future research is to enhance the quality of the clusters by clustering also in the time domain. This would be quite helpful e.g. for the CHICKEN sequence where the difficult extreme poses could then be separated.

Additionally we like to extend our approach to also handle the interpolation between key frames of an predefined animation, which is useful to smooth the animation or for motion blur. This may be possible by interpolating the computed affine transformations similar to [Alexa02]. Another possibility would be to find the transformations directly for the interpolated mesh (as explained in Section 3.3.3), which should be fast enough to do it just before rendering. On-the-fly interpolation between frames would not be possible anymore with pre-build, static kd-trees – neither would be skinned meshes.

For skinned meshes, under-sampling of the pose space can lead to underestimation of the fuzzy boxes and consequently to rendering artifacts. Thus instead of sampling the pose space it might be desirable to find the extent of the fuzzy boxes analytically, or by using affine arithmetic. Nevertheless, in practice we noticed no such artifacts when using a moderate number of samples (i.e. one sample per key frame, see also Table 3.1).

One direction for future work is the investigation of ray tracing even more general types of animations, such as blending meshes or meshes deformed by physical simulations. For example, ray tracing the cloth worn by a game character is not yet possible with our method.

3.7 Conclusion

We have presented a novel algorithmic approach to ray trace predefined animations and skinned meshes efficiently. We use a small top-level kd-tree and either automatically determined clusters or use the bone transformations to account for the dominating deformations of the mesh. As the residual motion of triangles can be handled by fuzzy kd-trees, we avoid almost completely the costly reconstruction of acceleration structures. As a result, we are able to achieve interactive frame rates even for high polygonal models.

Figure 3.10: *Example images of the* BEN *animation with textures and shadows. Enabling shadows we ray trace this detailed animation with a frame rate of about 2.2 fps using our fuzzy kd-trees compared to 4.1 fps when a single mesh is ray traced with a static kd-tree.*

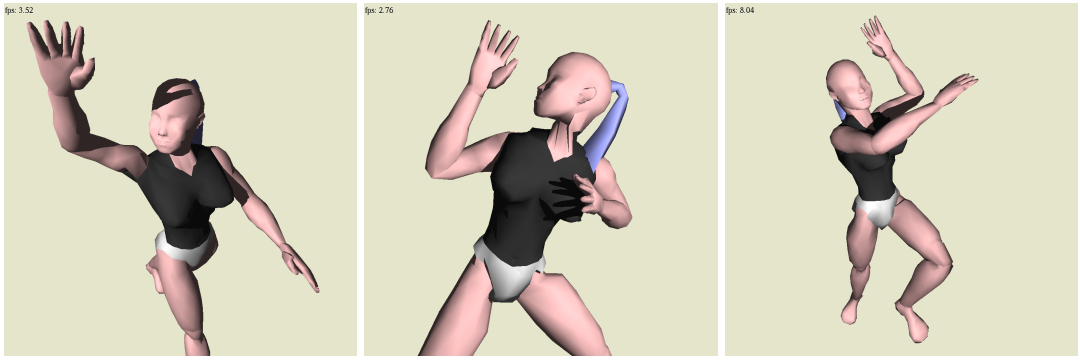


Figure 3.11: *CALLY featuring self shadowing in different poses, ray traced at interactive frame rates on a single 2.4 GHz Opteron CPU while supporting interpolation between poses with Cal3D.*

Chapter 4

Streamed Binning

In this chapter we present a new approach to construct high-quality kd-trees and BVHs. With *streamed binning* we achieve a much faster construction of these acceleration structures without sacrificing their culling performance for ray tracing. After reviewing traditional construction algorithms we show in detail how to apply streamed binning to the construction of kd-trees as well as to BVHs. Finally, we evaluate, compare and discuss our proposed algorithms.

4.1 Introduction and Related Work

Our goal is to construct acceleration structures from scratch every frame without any pre-knowledge of the scene. Consequently, the proposed algorithms are very flexible and do not imply any restrictions on the dynamic behavior of the scene: Primitives can be added or removed between frames, the connectivity as well as the topology can be changed, and primitives such as triangles can undergo completely random motion.

Historically, research of acceleration structures for ray tracing has concentrated on improving their culling performance, i.e., how well they can exclude primitives (in most cases triangles) from being tested for intersection with a ray. Kd-trees are already known for their protruding performance [Havran01], but also BVHs gained popularity due to their flexibility and their competitive performance [Wald07b]. Thus, we focus on these two acceleration structures in this chapter.

Key to efficient kd-trees and BVHs for ray tracing is the usage of the surface area heuristic (SAH) [MacDonald90] to guide their construction. Unfortunately, SAH-based construction of these high-performance acceleration structures is quite slow. This is not a problem for static scenes, as here the construction can be seen as a preprocess and its time is amortized over many rendered frames. For dynamic scenes, however, this is not the case anymore. Consequently, we look into methods to speed-up the build-

process and the SAH evaluation. Because of the importance of the SAH we discuss it in detail in Section 4.1.1.

An $\mathcal{O}(n \log n)$ algorithm for building kd-trees with a cost function (for photon mapping) is sketched in [Wald04] – this is the theoretical lower bound for comparison-based sorting algorithms. For a detailed analysis and a derivation of SAH-based kd-tree construction see [Wald06a] by Wald and Havran. Because our improvements are based on their algorithm we will also review it in Section 4.1.2.

Havran et al. [Havran06] analyzed the problem of fast construction of different spatial hierarchies. They also proposed a SAH-based hybrid acceleration structure that can be constructed quickly by exploiting the discretization of the whole 3D domain.

Hunt et al. [Hunt06] also aim at fast construction of kd-trees. By adaptive subsampling they approximate the SAH cost function by a piecewise *quadratic* function. Their SIMD scanning approach has complexity $\mathcal{O}(n \cdot k)$, with n being the number of primitives of the current node and k being the samples to take. That is why they can afford only a relative small number of samples (16) per split. Although we approximate the cost function only by a piecewise *linear* function, the degradation in quality of our constructed kd-trees can be neglected as we can use many more samples (1024), because streamed binning has complexity $\mathcal{O}(n+k)$ for finding a split.

Concurrently to our work, Wald [Wald07a] also applied streamed binning to BVHs. Wald tries only the major axis with only 16 bins in search for the split location, whereas we look at all three axes with up to 256 bins each. Also, we need at most $^2/_3$ of the memory bandwidth because we calculate the AABBs of the primitives from their centroid and extents on the fly instead of storing the AABBs *and* the centroids. Nevertheless, the resulting construction speeds of both implementations are comparable.

A memory efficient variant to split the triangle list during tree construction was proposed by Wächter and Keller [Wächter06]. They also report build times orders of magnitude faster than [Wald06a] by not using the costly SAH at all – consequently the resulting trees are not as efficient for ray tracing.

Hunt et al. [Hunt07] highlighted the benefits of closely coupling the scene graph and the acceleration structure for the quick and lazy construction of kd-trees, a strategy used by the Razor ray tracing system [Djeu11]. By exploiting that a reasonable scene graph provides geometry that is pre-sorted they can lower the asymptotic complexity of kd-tree construction to $\mathcal{O}(n)$. A further significant speed-up can be achieved when only these parts of the kd-tree are built that are actually needed.

A different class of approaches to accelerate tree construction is to parallelize the build and to utilize the high compute power of GPUs, which was firstly demonstrated by Zhou et al. [Zhou08]. They build kd-trees in breath-first order and according to cost functions in the lower levels of the tree.

Instead of a top-down construction, Lauterbach et al. [Lauterbach09] first sort the primitives in 3D according to their Morton code. They then build the tree bottom-up by stepwise clustering the primitives with group-wise identical bits in their morton code. This linear bounding volume hierarchy (LBVH) approach corresponds to splitting in middle and thus results in poor BVHs for ray tracing. Nevertheless, construction performance is very fast, also thanks to available parallel sorting primitives on the GPU. Lauterbach et al. also describe another GPU-based construction technique which uses SAH, resulting in good BVHs, but construction time suffers dramatically. Finally, they combine both methods to build only slightly inferior BVHs at reasonable fast speed.

A hierarchical version of LBVH, called HLBVH, was proposed by Pantaleoni and Luebke [Pantaleoni10]. They typically also apply SAH in top levels, resulting in BVHs with only moderate traversal performance loss (5–20 %). A simpler and even faster implementation of HLBVH was later presented by Garanzha et al. [Garanzha11a].

Garanzha et al. [Garanzha11b] also present an SAH-based BVH construction algorithm for the GPU with *linear* asymptotic complexity. As an auxiliary data structure they build a grid over the primitives and then hierarchically add the primitive-counts of neighboring cells in a mipmap-like manner. For each split they operate on one level of the grid and try at most 21 potential partitions (7 splits per dimension), taking the one with lowest SAH cost. As this is just a coarse approximation of the SAH, the resulting BVHs are up to 50% slower than BVHs build with the full SAH evaluation.

Quite astonishingly, Karras [Karras12] demonstrated that actually *each* node of a tree can be constructed in parallel, independent of all other nodes. Unfortunately, this finding resulted in no practical performance improvement.

So far, these GPU methods focus on parallelization, and need to trade tree quality for build performance. In contrast, Wu et al. [Wu11] fully evaluate SAH during kd-tree construction on the GPU, i.e. SAH is computed at all levels of the tree.

All these GPU-based construction algorithms are one to two orders of magnitude faster than our proposed algorithms for the CPU, which is relativized by the fact that GPU hardware offers one order of magnitude more compute power. But these GPU-based methods also share the disadvantage that all the to-be-processed geometry needs to fit into the rather small GPU memory. Thus, for large scenes such as the BOEING 777 (see Section 4.3) with its 350 million triangles there is currently no alternative to fast CPU-based construction algorithms.

The limited memory of GPUs is addressed by Hou et al. [Hou11] by proposing an out-of-core algorithm, where only parts of the BVH are processed at a time on the GPU. With this technique they can build BVHs for models of up to 20 million triangles. Although this is an order of magnitude larger than previously possible on the GPU, it is still far from what can be handled on the CPU. Additionally, the streaming of geometry to, and resulting BVH nodes from the GPU reduces the construction performance

to the point where it is not significantly faster anymore than CPU-based construction methods.

There have also been several attempts to parallelize tree construction on the CPU – with limited success in terms of scalability. Shevtsov et al. [Shevtsov07] report only 4× speedup on a 8 core machine for parallel kd-tree construction. Similarly, Wald [Wald07a] state poor scalability beyond two CPU cores for parallel BVH construction. More recently, Choi et al. [Choi10] achieve just about 8× speedup on a 32 core workstation for kd-tree construction. The reasons for this poor scalability behavior are not fully clear. One reason may be that there are still significant serial parts in the construction algorithm, e.g. that the initial sort of the primitives was not parallelized. Another reason could be limitations of the memory subsystem, i.e. that the bandwidth of the main memory is to small to sufficiently serve all CPU cores.

However, the currently best results were reported by Wald [Wald12] for the Intel Many Integrated Cores (MIC) architecture with near linear speedup for reasonable large scenes. Also, absolute construction performance is very competitive to GPU-based construction algorithms while simultaneously providing quite high-quality BVHs based on SAH. The carefully optimized implementation is based on several concepts: bounding boxes are quantized during build to minimize the cache footprint and the required memory bandwidth, data structures and kernels are tailored for the MIC instruction set, and a lightweight tasking system reduces synchronization overhead. Unfortunately, the used MIC architecture prototype board has quite limited memory and thus the test scenes were rather small.

4.1.1 The Surface Area Heuristic

The surface area heuristic (SAH) [Goldsmith87, MacDonald89, MacDonald90, Subramanian90] estimates the ray tracing performance of a given acceleration structure. This global cost C_{Tree} can be computed for a given constructed kd-tree or BVH *Tree* as

$$C_{Tree} = K_{Trav} \sum_{N \in Nodes} \frac{\mathrm{SA}(V_N)}{\mathrm{SA}(V_S)} + K_{Isec} \sum_{L \in Leaves} \frac{\mathrm{SA}(V_L)}{\mathrm{SA}(V_S)} n_L, \tag{4.1}$$

where $\mathrm{SA}(V)$ is the surface area of the axis-aligned bounding box (AABB) V, V_S is the AABB of the whole scene, K_{Trav} and K_{Isec} are constants for the cost of a traversal and an intersection step, respectively, and n_L is the number of primitives in leaf L.

The derivation of this statistical cost-model is based on the assumption that rays can be seen as infinite lines that are uniformly distributed. Then, the probability P of a ray hitting a (convex) volume V is proportional to the surface area SA of that volume

(see e.g. [Santalo02]). In particular, if a ray is known to hit a volume V_S, the probability of hitting a sub-volume V_S is

$$\mathrm{P}(V|V_S) = \frac{\mathrm{SA}(V)}{\mathrm{SA}(V_S)} \tag{4.2}$$

Actually, these assumptions are idealized and do not hold in practice: rays are often not uniformly distributed and are not infinite lines, but usually start within the scene and terminate when hitting a primitive. Consequently, several researchers addressed these shortcomings and proposed improved heuristics with more realistic assumptions. Havran developed a general cost metric [Havran01], which also considers ray and geometry interaction. Ize and Hansen also proposed a modified SAH accounting for ray termination, called RTSAH [Ize11]. They successfully use RTSAH to guide the traversal order of occlusion rays. Others studied the construction of trees tailored for non-uniform ray distributions [Bittner09, Feltman12, Vinkler12]. Further improvements to the SAH are described in [Hunt08, Fabianowski09]. Note however, despite its theoretical shortcomings the SAH works surprisingly well in practice and we will therefore work with the original heuristic.

High-performance kd-trees and BVHs are built according to the SAH by minimizing their expected cost (4.1) for ray tracing. However, evaluating all potential tree configurations to solve this global optimization problem is impractical even for small scenes. Fortunately, a *local greedy approximation* for a recursive top-down construction works well for kd-trees and BVHs [Wald07b]. For each node N to be split into two child nodes N_l and N_r the cost C_{Part} of each potential partition is computed according to

$$\begin{aligned} C_{Part} &= K_{Trav} + K_{Isec}\left[n_l\,\mathrm{P}(N_l|N) + n_r\,\mathrm{P}(N_r|N)\right] \\ &\overset{(4.2)}{=} K_{Trav} + \frac{K_{Isec}}{\mathrm{SA}(N)}\left[n_l\,\mathrm{SA}(N_l) + n_r\,\mathrm{SA}(N_r)\right], \end{aligned} \tag{4.3}$$

where n_l and n_r are the number of contained primitives in the respective child nodes, i.e. the children are treated as leaves and their costs are weighted by the probability of hitting them. We take that partition that has minimal local cost C_{Part}, or, if the minimal C_{Part} is greater than the cost of not splitting at all ($K_{Isec}\cdot(n_l+n_r)$), a leaf is created.

For a kd-tree the minimum of C_{Part} can only be realized at a split plane position where primitives start or end, thus the bounding planes of primitives are taken as potential split plane candidates. Furthermore, C_{Part} depends on the surface area SA of the children – which can be directly computed – and the primitive count of the children – which is the hardest part to compute efficiently.

For a BVH the partitions are usually also found by trying axis-parallel split planes and comparing the centroids of the primitives against the split plane [Wald07b]. However, this way many of the $2^{n-1}-1$ possible partitions of n primitives for a BVH are not tested. Notably, although considering these additional partitions can indeed result in lower SAH cost per local greedy split [Popov09], Popov et al. found that the complete BVH has higher SAH cost and performs worse compared to a BVH built with just using the axis-parallel split planes.

4.1.2 Conventional SAH-based Tree Construction

We will revisit the conventional SAH-based construction of kd-trees [Wald06a] and BVHs [Wald07b]. Both construction algorithms are sufficiently similar so we describe them together and only mention differences where appropriate. Also note that both construction algorithms already have asymptotic complexity of $\mathcal{O}(n \log n)$, which is optimal for comparison-based sorting algorithms.

The trees are typically built in a top-down fashion by recursively splitting the tree into two sub-trees, or correspondingly, partitioning a node into two children. The expected ray tracing cost is calculated according to the SAH for possible split configurations, and the split with minimal cost is taken. The recursion stops – and the current node is made a leaf – if the minimal splitting cost is higher than the cost of not splitting.

The cost function (4.3) is evaluated by sweeping the bounding box of the current node N with three planes perpendicular to the x-, y- and z-axis. For kd-trees it has been shown that the cost function is linear between the boundaries of the AABBs of the primitives inside N [Havran01]. Hence, only these locations, or *events*, need to be considered when searching for the minimum of the cost function. For BVHs traditionally the centroids of the primitives are taken as potential split events [Wald07b].

The cost function is evaluated incrementally during the sweep by keeping track of the number of primitives belonging to the left and right child for each split event, which requires the events of each dimension to be sorted. For a kd-tree the surface area of each potential child can directly be computed from the position of the split plane. For a BVH this is not so easy, because during the sweep the AABB of each potential child can change in each dimension. Therefore, actually two sweeps are needed: the first one incrementally enlarges one child by adding each primitive step-by-step, calculates its surface area and temporarily stores the result; the second sweep in the opposite direction incrementally enlarges the other child and computes the cost function.

Sorting the events for each split results in a $\mathcal{O}(n \log^2 n)$ construction algorithm. To achieve the better time complexity of $\mathcal{O}(n \log n)$, the event lists are sorted only *once* at the beginning of the construction. During the split the events are then sifted to the two new nodes, keeping their order. To do so, the primitives are first classified as belonging

to the left or right with respect to the chosen event location and dimension, such that the events of the other both dimensions can be sifted accordingly.

4.2 Fast Construction of Acceleration Structures

Although reaching the optimal asymptotic complexity of $\mathcal{O}(n \log n)$ the classic construction algorithms for kd-trees and BVHs have several main drawbacks: First, the initial sort of the events is expensive, already having a time complexity of $\mathcal{O}(n \log n)$. Thus, only partially building the tree in a lazy construction scheme is not reasonable. Second, the sifting phase generates random memory accesses, because in general the events of the same primitive have a difference place in the sorted event lists per dimension. This is in particular problematic for the first splits for larger (real-world) scenes, where these random memory accesses cannot be caught by the cache hierarchy. Third, for each split many passes over the data of a node are performed (SAH evaluation, classification, sifting), which considerably consumes the bandwidth to the slow main memory (again, at least for the first splits of larger scenes).

Because we cannot hope to find tree construction algorithms with lower time complexity, we need to concentrate on lowering the constants, i.e. doing less work for each split and in particular optimizing memory access patterns. Thus we look again at the actual problem of efficiently evaluating the SAH cost for a partition (4.3): at each potential split location we need the number of the primitives that will go to either side[1]. If we want to have these counts for all ($\mathcal{O}(n)$ many) events then they must be available in constant time, leading to the incremental counting algorithm in sorted event list of the classical approach. However, if we evaluate Equation (4.3) only at a *fixed number of locations*, counting can take $\mathcal{O}(n)$ time and we still achieve optimal time complexity for the complete tree construction.

This is the core idea of our new approach: we approximately find the minimum of the SAH cost by sub-sampling this function at fixed locations. Thus we save computations due to less SAH evaluations. To get the primitive counts we iterate once per split over the primitives and sort them into bins in $\mathcal{O}(n)$ time. The gathered information in the bins is then used to reconstruct the primitive counts on both sides of each border between the bins, and thus to compute the SAH cost function at each border location. This way we also skip the initial sort of the events and only access the memory for the bulk data of primitives in a regular, streamed manner.

In the following we detail how this general idea is applied and specialized to the fast construction of kd-trees and BVHs, as there are some differences in binning, SAH

[1] We ignore the surface area for the moment.

evaluation, and splitting.

4.2.1 Streamed Binning for KD-Trees

For a kd-tree primitives that straddle the split plane[2] of a node need to be referenced in both children, and consequently they also need to be considered twice when calculating the SAH cost function. More precisely, all primitives that *start* left of the split position will go to the left child, and all primitives that *end* right of the split position will go to the right child. Therefore, each bin has two entries: one counter for the start events and one counter for the end events that fall into this bin.

Binning is done for all three dimensions at once: the AABBs of the primitives are streamed in, the indices of the bins are found from the minimum (for the start counters) and the maximum extents (for the end counters) of the AABB, and the corresponding counters of the bins are incremented.

After binning the bins are processed from left to right to find the approximate minimum of the SAH cost function. The number of primitives going left (n_l) and going right (n_r) are computed by a prefix sum. At the beginning $n_l = 0$ and n_r is initialized with the total number of primitives of the processed node. Then, from bin to bin n_l is updated by adding the start counter whereas n_r is updated by subtracting the end counter. Updating the surface areas of the left and right child is easy as they only change by a constant from bin to bin.

We use vector SIMD instructions to perform binning and SAH evaluation for all three dimensions in parallel. For binning we use 1024 uniformly distributed sample locations per dimension.

When the split with minimal cost is found the AABBs of the primitives are sifted to the left and right child by comparing them to the split location, AABBs that go to both child nodes are clipped. During sifting we already perform binning for the two children, further reducing memory bandwidth. The child nodes are then processed recursively.

When a node is small enough we revert to the classic construction approach, because binning becomes inefficient when the number of primitives gets close to the number of bins. But now the working set fits into the cache and thus the random memory access and bandwidth disadvantages of the classic approach do not hurt anymore. Being in the cache now also offers the option to use the fast $\mathcal{O}(n)$ radix sort algorithm [Knuth98, Sedgewick98] to initially sort the events of the node, because radix sort also accesses memory in random order.

[2]Note that we skip a potential optimization called *perfect split* [Havran01] for the straddling primitives, which would require costly clipping operations and bounding box recomputations.

4.2.2 Streamed Binning for BVHs

To enumerate partitions of a node during BVH construction we follow [Wald07b] and use a set of uniformly distributed, axis-aligned planes to distribute the primitives by means of their centroids. For each potential partition we need to compute (4.3), hence we need to know the primitive counts *and* the surface areas of both children.

Therefore, each bin consists of an AABB and a counter. The primitives are represented by the centroid and the extent of their AABBs. For each primitive we compute the indices of the bins of all three dimensions from its centroid in SIMD. Then, the counters of all three bins are incremented, and their AABBs are enlarged with the primitive's AABB using SIMD min/max operations. Note that accumulating the *extent* in the bins is necessary as well, because – unlike kd-trees – the split plane location alone is not sufficient to compute the surface areas of the child nodes, because the AABBs of the children can shrink in all three dimensions.

We enhance the resolution of the binning by uniformly distributing the bins over the current interval of all the *centroids* rather than over the current bounding box of the primitives. This is especially important when there are large primitives.

After binning we evaluate Equation (4.3) with two passes over the bins: In the first pass from left to right we compute n_l and $\mathrm{SA}(N_l)$ at the borders of the bins by accumulating the counters and by successively enlarge the AABBs of the bins. In the second pass from right to left we reconstruct n_r and $\mathrm{SA}(N_r)$, and finally find the index i_{min} and dimension of the bin that has minimal cost C_{Part} at its left border.

Computing the split plane position from i_{min} turned out to be surprisingly difficult. Because of floating point precision problems we cannot just invert the linear function used during binning. An inaccurate split plane can not only lead to sub-optimal partitions. In the worst case, an inaccurate split plane can even lead to an invalid partitions (one child is empty) if the split plane is computed to be completely on one side of all centroids. Using double precision only reduces the chances of invalid partitions but does not solve the problem. Our solutions is to not compute the splitting plane from i_{min} at all, but to keep track of the centroids during binning. Therefore each bin additionally stores the minimum of the coordinates of all centroids that fell into it. Using the centroid minimum of bin i_{min} as split plane location then ensures consistent and robust partitioning.

As with kd-tree construction we minimize the required memory bandwidth by performing the binning in all three dimensions for both children during the split of the parent node. Because primitives are referenced exactly once in a BVH we can actually perform the sifting of the AABBs *in place*. Also for BVH construction we extensively use SIMD operations to exploit instruction level parallelism of modern CPUs, working on all three dimensions at once during binning and during SAH evaluation.

The number of bins is a crucial parameter controlling the construction speed and accuracy. The more bins there are, the more accurate is the sampling of the SAH cost function, but the more work has to be done during calculation of the SAH function from the binned data (the binning steps themselves are independent of the number of the bins). The number of bins per dimension should be chosen such that the binning data of both children still fits into the L1 cache.

Additionally, binning becomes inefficient if the number of bins is close to the number of to-be-binned primitives. Instead of switching to the classical construction algorithm as done for kd-trees we adaptively choose the number of bins k per dimension linearly depending on the number of primitives n and bin-ratio r as $k = n/r$ and clamp it to $[k_{min}, k_{max}]$. We experimented with different parameter sets representing a trade-off between speed and accuracy. The `default` settings are $k_{max} = 128$, $k_{min} = 8$, and $r = 6$. The `fast` settings are $k_{max} = 32$, $k_{min} = 4$, and $r = 16$.

4.3 Results and Discussion

We evaluated our construction algorithms with a variety of scenes (see Figure 4.1). These scenes differ not only in size (number of triangles), but also in regularity:

On the one hand, there are the BUNNY, BUDDHA, BLADE, and THAI STATUE models from the Stanford 3D Scanning Repository, which were acquired using a 3D scanning process. That is why these four scenes have very uniformly distributed triangles of similar size. Additionally, the triangles are already stored in a spatial order.

On the other hand, the HAND, FAIRY FOREST, CONFERENCE room, the BEETLE model, SODA HALL, POWER PLANT, and the BOEING 777 were designed by hand or with CAD tools. Except for the HAND these scenes have quite differently sized triangles that are irregularly distributed. Thus the SAH should be effective in guiding the acceleration structure construction.

For our measurements we used a workstation with two AMD 2.6 GHz dual-core Opterons as well as an Intel 2.4 GHz Core 2 system.

To evaluate the performance of our streamed binning construction algorithms for kd-tree and BVH we compare them to the conventional construction algorithms in terms of the time needed to build an SAH-based acceleration structure for the test scenes and in terms of quality of the resulting acceleration structures.

4.3.1 KD-Tree Construction Speed

Table 4.1 reveals that several factors influence the benefit of streamed binning for kd-trees. First, the number of triangles determines the memory requirements and thus the

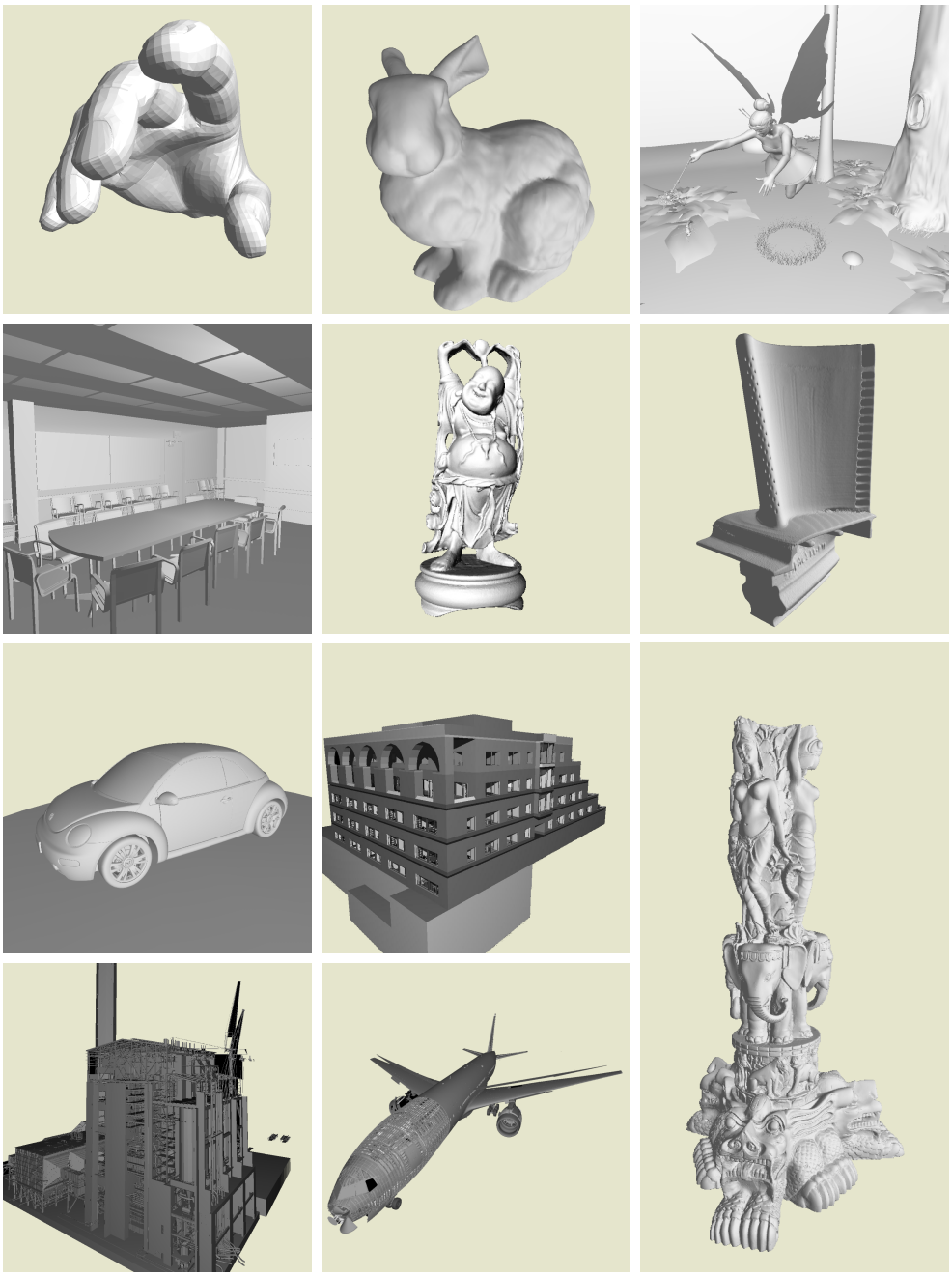


Figure 4.1: *The scenes used for testing and evaluation of our streamed binning construction algorithms. In reading order:* HAND, BUNNY, FAIRY FOREST, CONFERENCE, BUDDHA, BLADE, BEETLE, SODA HALL, THAI STATUE, POWER PLANT, *and* BOEING 777.

cache efficiency. For larger scenes it is likely that the irregular and incoherent memory accesses of the classic construction cannot be compensated by the caches. Therefore, the improvement using our streamed binning construction over the conventional algorithm is higher for larger scenes.

Second, if the triangles of a scene are already partially sorted (as is the case with the scanned models), then the locality of memory accesses is improved in favor for the conventional kd-tree construction. To measure the impact of this pre-sorting we also tried our kd-tree construction algorithm on the same scenes, but with randomly shuffled triangles. With the shuffled scenes the streamed binning approach clearly outperforms the conventional construction with a speedup of up to 50%. Furthermore, we provide strong evidence that our streamed binning construction is independent of the order of the input data, because shuffling the triangles has hardly any effect on the construction times, even for large scenes. This is important because we cannot (and should not) rely on any meaningful sorting for dynamic scenes.

Finally, we see that the construction times also depend on the type of the scene and the distribution of the triangles. For example the BLADE and the BEETLE have roughly the same number of triangles. Still, to build a kd-tree for the BEETLE takes about twice as long, because of the more irregularly sized triangles.

Additional to these measurements we also profiled the different parts of our construction algorithm. Building the many smaller nodes that are lower in the tree consumes most of the construction time, and unfortunately, the streamed binning approach does not help anymore for these nodes. Furthermore, we observe that the time spent in sifting the events to the left and right children already dominates the construction time.

4.3.2 BVH Construction Speed

In Table 4.2 we compare absolute construction times of the conventional BVH construction algorithm (from [Wald07b] and also from an own implementation) with the results of our new streamed binning approach. As can be seen, streamed binning is *one order of magnitude faster*, at almost the same BVH quality. Using the `fast` parameter settings (see Section 4.2.2) BVH construction is about 20% faster than using the `default` settings, at the cost of slightly decreased BVH quality. Even for such a large scene as the 350 million triangle BOEING 777 we show that with our streamed binning construction algorithm we can produce a high quality BVH in less than ten minutes.

Streamed binning for BVH construction is more computational demanding than for kd-tree construction, because we need to keep track not only of the primitive counts, but also of the surface areas of the children. Additionally, the surface area cannot be incrementally computed, because the AABBs may have changed in all three dimensions.

scene	#triangles	Conventional Construction time (shuffled)		Streamed Binning KD-Tree Construction time (shuffled)		speedup (shuffled)		quality
HAND	17,135	109 ms	112 ms	103 ms	106 ms	5.5%	5.7%	100.0%
BUNNY	69,451	611 ms	622 ms	513 ms	520 ms	19%	19%	99.0%
FAIRY FOREST	174,117	1.32 s	1.45 s	1.15 s	1.18 s	15%	23%	98.9%
CONFERENCE	282,664	1.55 s	1.83 s	1.41 s	1.48 s	9.5%	24%	98.4%
BLADE	1,765,388	7.52 s	10.4 s	7.60 s	8.04 s	-1.1%	29%	99.2%
BEETLE	1,873,389	18.8 s	22.0 s	15.3 s	15.6 s	23%	41%	97.9%
THAI STATUE	10,000,000	99.6 s	123 s	80.8 s	83.1 s	23%	48%	98.8%

Table 4.1: *Comparison between the conventional kd-tree construction algorithm and our new streamed binning construction algorithm with respect to building time and quality of the resulting kd-tree, using one 2.6 GHz Opteron CPU. The larger the scene, the higher the speedup using streamed binning construction because for the conventional build the working set does not fit into the caches anymore. Additionally, the conventional build heavily depends on pre-sorted input whereas streamed binning construction is largely independent of the ordering of the triangles. The increased expected cost by only maximal 2% due to the approximative evaluation of the SAH cost function with binning can be well tolerated.*

Nevertheless, constructing an SAH BVH with streamed binning can still be faster than constructing an SAH kd-tree for the same scene: Because a BVH does not split primitives, less nodes need to be created. Also, a BVH is usually not as deep as a kd-tree (using the same SAH termination criterion), because a BVH node bounds in three dimensions whereas a kd-tree node bounds only in one dimension. Thus, the number of splits and binning steps to be performed during construction is lower for a BVH.

These considerations are backed up by our measurements when comparing the times of Table 4.1 and Table 4.2. Building BVHs with streamed binning is several times faster than building kd-trees, whether the kd-trees are build with streamed binning or conventionally.

We also observe that streamed binning is much more beneficial for BVH construction than for kd-tree construction. For BVHs the evaluation of the SAH cost function is dominating, in particular the computation of the surface area. With binning we significantly reduce this part of the construction. Another reason may also be that for BVHs we do not switch to conventional construction once a node is below a certain size. Instead we cope with the reduced efficiency of binning when there are only few primitives by accordingly reducing the numbers of bins.

4.3.3 Approximative Cost Function Evaluation

We also estimated the approximation error introduced by sub-sampling the SAH cost function in the streamed binning splits, because this influences the quality of the built kd-trees and BVHs. Therefore, Table 4.1 and Table 4.2 also provide the expected ray tracing cost according the SAH (4.1) for the constructed acceleration structures. Additionally, we also evaluated the average intersection cost for a ray by intersecting uniformly generated rays with the scene using the generated kd-trees. As the expected costs and the measured intersection costs were strongly correlated we excluded these latter numbers.

For our test scenes the sub-sampling approach during streamed binning construction hardly decreased the quality of neither the kd-trees nor of the BVHs. With at most 2.2% the approximation error is very low. Although the `fast` settings for BVH construction gain about 20% construction speed they can also reduce the BVH quality in extreme cases as for the CONFERENCE scene from 0.6% to 7.5%.

Interestingly, for some scenes the binning approximation of the SAH cost function results in even *better* BVHs (quality $> 100\%$ in Table 4.2) than when exactly evaluating Equation (4.3). This is a strong confirmation that the local greedy SAH function is exactly that, a local greedy optimization, failing to provide the global minimal SAH cost (4.1).

		Conventional Construction	Streamed Binning BVH Construction				
		[Wald07b]	Our Implementation (2.4 GHz Core 2)				
scene	#triangles	(2.6 GHz Opteron)	exact SAH	binning	quality	fast binning	quality
BUNNY	69,451	—	168 ms	48 ms	99.8%	37 ms	98.9%
FAIRY FOREST	174,117	2.8 s	0.47 s	0.12 s	100.2%	0.10 s	98.8%
CONFERENCE	282,641	5.06 s	0.80 s	0.20 s	99.4%	0.15 s	92.5%
BUDDHA	1,087,716	20.8 s	4.38 s	0.84 s	100.0%	0.66 s	98.9%
SODA HALL	2,169,132	53.2 s	8.78 s	1.59 s	101.6%	1.28 s	103.5%
POWER PLANT	12,748,510	—	119 s	8.1 s	100.5%	6.6 s	99.4%
BOEING 777	348,216,139	—	5605 s	667 s	98.1%	572 s	94.8%

Table 4.2: *Comparing the construction performance for BVHs using different construction algorithms on similar hardware. Due to its huge size the* BOEING 777 *was measured on a 2.0 GHz Opteron with 64 GB RAM, of which 35 GB were consumed during construction. The reported quality of our proposed streamed binning BVH construction is measured in SAH cost C_{Tree}* (4.1) *and is relative to the exact SAH evaluation. Streamed binned construction for BVHs is both very fast and accurate.*

4.4 Conclusion

We presented a new construction method for kd-trees and BVHs that is based on streamed binning to approximately evaluate the SAH cost function when searching for the best split position. Due to the very low approximation error the resulting kd-trees and BVHs are of high quality for ray tracing.

Because BVHs can be built much faster than kd-trees BVHs should be used for ray tracing dynamic scenes: With a construction speed of about 2 million triangles per second we can already build a BVH from scratch every frame to handle arbitrary dynamic changes in smaller to medium sized scenes.

The high construction performance in time and quality and the easy implementation make the streamed binning algorithm a prime candidate for BVH construction for ray tracing. It can also efficiently handle huge scenes on the CPU.

Currently we place the bins uniformly, which helps when computing the indices of the bins form the coordinates of the AABBs. However, it is also possible to place the samples non-uniformly according to a simple-to-evaluate function to increase the sampling density, for example, near the spatial center of a node.

Chapter 5

Conclusion

The advances in ray tracing performance in the last decade – reaching interactive frame rates for non-trivial scenes – roused the desire to also ray trace *dynamic* scenes. Changing scenes, however, pose a great challenge to ray tracing, because the precomputed auxiliary data-structures to accelerate ray tracing get invalidated by changes of the geometry. In this thesis we reviewed and discussed several approaches to deal with this problem, the problem of ray tracing dynamic scenes. As main contributions we developed and described two of these approaches, namely *motion decomposition* and *streamed binning*.

The motion decomposition approach aims to avoid many invalidations and consequently recomputations of acceleration structures upon changes of the scene. The idea is to segment the scene into parts, to separate large-scale motion between these parts from residual motion within each part. These parts of the scene are organized into a small top-level kd-tree, which is quickly rebuilt every frame to accommodate for the large-scale motion. In contrast, the residual motion is handled by fuzzy kd-trees that are robust against small changes in geometry and thus stay valid and do not require reconstruction. Motion decomposition assumes locally coherent motion. Additionally, the amount of motion needs to be known in advance.

We demonstrated how to apply this approach to two scenarios: to predefined animations and to skinned meshes. To find the decomposition of the scene for predefined animations we developed a clustering algorithm based on Lloyd relaxation. It minimizes the residual motion in each produced part of the scene and thus optimizes ray tracing performance. For skinned meshes the decomposition exploits the information of the bones that drive the skinning animation of the mesh. As a result, with motion decomposition we can ray trace these types of animations at interactive frame rates.

Although is was not the original goal of the motion decomposition framework we could also combine it with BVHs. BVHs can be refitted to adapt to the residual motion and they could thus replace the fuzzy kd-trees. The clustering algorithms of motion

decomposition would still adapt to the dominating deformations of the scene and would thus find good structures of BVHs. This is crucial, because using BVH refitting alone quickly leads to deteriorated trees.

The streamed binning approach aims to reduce the time required for building the acceleration structures, so that they can be reconstructed fast enough after changes to the scene, ultimately several times a second. Streamed binning quickens the construction of kd-trees and BVHs by approximately finding the minimum of the SAH cost function for the best split position by reducing the number of tested split positions. Additionally, we avoid the previously necessary initial sort of the primitives, leading to fast, streamed memory accesses.

The introduced approximation error is very low, such that the resulting kd-trees and BVHs are of high quality for ray tracing. Therefore, streamed binning can replace the conventional construction algorithms in most usage scenarios. We showed that the construction of BVHs benefits much more from our new technique than the construction of kd-trees. BVHs can already be built fast enough from scratch every frame to handle arbitrary dynamic changes in smaller to medium sized scenes. For large scenes streamed binning greatly reduces the time to image, but is alone not yet fast enough for interactive use and should be complemented with other techniques such as a two-level acceleration structure scheme if necessary.

As we have shown in this thesis there are several different approaches to ray trace changing scenes, suited for different requirements and scenarios.

List of Publications

[Wald09] *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. *Computer Graphics Forum*, 28(6):1691–1722, 2009.

[Günther07] *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*, pages 113–118, September 2007.

[Wald07] *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 *STAR Proceedings of Eurographics 2007*, pages 89–116. The Eurographics Association, September 2007.

[Popov07] *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):415–424, September 2007. (Proceedings of Eurographics).

[Popov06] *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.

[Wald06] *Ingo Wald, Andreas Dietrich, Carsten Benthin, Alexander Efremov, Tim Dahmen, Johannes Günther, Vlastimil Havran, Hans-Peter Seidel, and Philipp Slusallek*. A Ray Tracing based Framework for High-Quality Virtual Reality in Industrial Design Applications. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 177–185, September 2006.

[Friedrich06] *Heiko Friedrich, Johannes Günther, Andreas Dietrich, Michael Scherbaum, Hans-Peter Seidel, and Philipp Slusallek*. Exploring the Use of Ray Tracing for Future Games. In *sandbox '06: Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames*, pages 41–50. ACM Press, 2006.

[Günther06b] *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).

[Günther06a] *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).

[Günther05] *Johannes Günther, Tongbo Chen, Michael Goesele, Ingo Wald, and Hans-Peter Seidel*. Efficient Acquisition and Realistic Rendering of Car Paint. In *Proceedings of 10th International Fall Workshop - Vision, Modeling, and Visualization (VMV) 2005*, pages 487–494, November 2005.

[Günther04] *Johannes Günther, Ingo Wald, and Philipp Slusallek*. Realtime Caustics using Distributed Photon Mapping. In *Rendering Techniques*, pages 111–121, June 2004. (Proceedings of the 15th Eurographics Symposium on Rendering).

[Wald04] *Ingo Wald, Johannes Günther, and Philipp Slusallek*. Balancing Considered Harmful – Faster Photon Mapping using the Voxel Volume Heuristic. *Computer Graphics Forum*, 22(3):595–603, 2004. (Proceedings of Eurographics).

Bibliography

[Áfra12] *Attila T. Áfra*. Incoherent Ray Tracing without Acceleration Structures. In Carlos Andujar and Enrico Puppo, editors, *EG 2012 – Short Papers*, pages 97–100. Eurographics Association, 2012.

[Aila09] *Timo Aila and Samuli Laine*. Understanding the Efficiency of Ray Traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics*, pages 145–149, 2009.

[Alexa00] *Marc Alexa and Wolfgang Müller*. Representing Animations by Principal Components. *Computer Graphics Forum*, 19(3):411–418, 2000.

[Alexa02] *Marc Alexa*. Linear Combination of Transformations. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 380–387, New York, NY, USA, 2002. ACM Press.

[Amanatides87] *John Amanatides and Andrew Woo*. A Fast Voxel Traversal Algorithm for Ray Tracing. In *Proceedings of Eurographics*, pages 3–10. Elsevier Science Publishers, Amsterdam, North-Holland, 1987.

[Anderson99] *E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen*. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, Third edition, 1999.

[Arvo87] *James Arvo and David Kirk*. Fast Ray Tracing by Ray Classification. *Computer Graphics (Proceedings of ACM SIGGRAPH)*, 21(4):55–64, 1987.

[Arvo88] *James Arvo*. Linear-time Voxel Walking for Octrees. *Ray Tracing News (available at `http://tog.acm.org/resources/RTNews/html/rtnews2d.html#art5`)*, 1(5), March 1988.

[Barequet01] *Gill Barequet and Sariel Har-Peled*. Efficiently Approximating the Minimum-Volume Bounding Box of a Point Set in Three Dimensions. *J. Algorithms*, 38:91–109, 2001.

[Benthin12] *Carsten Benthin, Ingo Wald, Sven Woop, Manfred Ernst, and William R. Mark*. Combining Single and Packet-Ray Tracing for Arbitrary Ray Distributions on the Intel MIC Architecture. *IEEE Transactions on Visualization and Computer Graphics*, 18(9):1438–1448, September 2012.

[Bittner09] *Jiří Bittner and Vlastimil Havran*. RDH: Ray Distribution Heuristics for Construction of Spatial Data Structures. In *Proceedings of the 2009 Spring Conference on Computer Graphics*, pages 51–58. ACM, 2009.

[Bittner13] *Jiří Bittner, Michal Hapala, and Vlastimil Havran*. Fast Insertion-Based Optimization of Bounding Volume Hierarchies. *Computer Graphics Forum*, 32(1):85–100, 2013.

[Cal3D] *Cal3D*. 3D Character Animation Library. `http://gna.org/projects/cal3d/`.

[Choi10] *Byn Choi, Rakesh Komuravelli, Victor Lu, Hyojin Sung, Robert L Bocchino, Sarita V Adve, and John C Hart*. Parallel SAH kD tree Construction. In *Proceedings of the Conference on High Performance Graphics*, pages 77–86. Eurographics Association, 2010.

[Cleary83] *John Cleary, Brian Wyvill, Graham Birtwistle, and Reddy Vatti*. A Parallel Ray Tracing Computer. In *Proceedings XI Association of Simula Users Conference*, pages 77–80, 1983.

[Cook84] *Robert Cook, Thomas Porter, and Loren Carpenter*. Distributed Ray Tracing. *Computer Graphics (Proceeding of ACM SIGGRAPH)*, 18(3):137–144, 1984.

[Dammertz08a] *Holger Dammertz, Johannes Hanika, and Alexander Keller*. Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays. *Computer Graphics Forum*, 27(4):1225–1234, 2008. (Proceedings of the 19th Eurographics Symposium on Rendering).

[Dammertz08b] *Holger Dammertz and Alexander Keller*. The Edge Volume Heuristic – Robust Triangle Subdivision for Improved BVH Performance. In *Proceedings of the 2008 IEEE Symposium on Interactive Ray Tracing*, pages 155–158. IEEE, 2008.

[Djeu11] *Peter Djeu, Warren Hunt, Rui Wang, Ikrima Elhassan, Gordon Stoll, and William R Mark*. Razor: An Architecture for Dynamic Multiresolution Ray Tracing. *ACM Transactions on Graphics*, 30(5):115:1–115:26, 2011.

[Du99] *Qiang Du, Vance Faber, and Max Gunzburger*. Centroidal Voronoi Tessellations: Applications and Algorithms. *SIAM Rev.*, 41(4):637–676, 1999.

[Eisemann07] *Martin Eisemann, Thorsten Grosch, Marcus Magnor, and Stefan Müller*. Automatic Creation of Object Hierarchies for Ray Tracing Dynamic Scenes. In Vaclav Skala, editor, *WSCG Short Papers Post-Conference Proceedings*, January 2007.

[Ernst07] *Manfred Ernst and Günther Greiner*. Early Split Clipping for Bounding Volume Hierarchies. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pages 73–78. IEEE, 2007.

[Ernst08] *Manfred Ernst and Günther Greiner*. Multi Bounding Volume Hierarchies. In *Proceedings of the 2008 IEEE Symposium on Interactive Ray Tracing*, pages 35–40. IEEE, 2008.

[Fabianowski09] *Bartosz Fabianowski, Colin Fowler, and John Dingliana*. A Cost Metric for Scene-Interior Ray Origins. In *Eurographics Short Papers*, pages 49–52, 2009.

[Feltman12] *Nicolas Feltman, Minjae Lee, and Kayvon Fatahalian*. SRDH: Specializing BVH Construction and Traversal Order Using Representative Shadow Ray Sets. In *Proceedings*

of the Fourth ACM SIGGRAPH / Eurographics conference on High-Performance Graphics, pages 49–55. Eurographics Association, 2012.

[Foley05] *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.

[Fuchs80] *Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor*. On Visible Surface Generation by a Priori Tree Structures. *Computer Graphics (Proceedings of SIGGRAPH '80)*, 14(3):124–133, 1980.

[Garanzha10] *Kirill Garanzha and Charles Loop*. Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing. *Computer Graphics Forum*, 29(2):289–298, 2010. (Proceedings of Eurographics).

[Garanzha11a] *Kirill Garanzha, Jacopo Pantaleoni, and David McAllister*. Simpler and Faster HLBVH with Work Queues. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, pages 59–64. ACM, 2011.

[Garanzha11b] *Kirill Garanzha, Simon Premože, Alexander Bely, and Vladimir Galaktionov*. Grid-based SAH BVH Construction on a GPU. *The Visual Computer*, 27(6-8):697–706, 2011.

[Georgiev08] *Iliyan Georgiev and Philipp Slusallek*. RTfact: Generic Concepts for Flexible and High Performance Ray Tracing. In *Proceedings of the 2008 IEEE Symposium on Interactive Ray Tracing*, pages 115–122. IEEE, 2008.

[Gersho91] *Allen Gersho and Robert M. Gray*. *Vector quantization and signal compression*. Kluwer Academic Publishers, Norwell, MA, USA, 1991.

[Glassner84] *Andrew S. Glassner*. Space Subdivision For Fast Ray Tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, October 1984.

[Goldsmith87] *Jeffrey Goldsmith and John Salmon*. Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics and Applications*, 7(5):14–20, May 1987.

[Havran01] *Vlastimil Havran*. Heuristic Ray Shooting Algorithms. PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, 2001.

[Havran06] *Vlastimil Havran, Robert Herzog, and Hans-Peter Seidel*. On Fast Construction of Spatial Hierarchies for Ray Tracing. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 71–80, September 2006.

[Hou11] *Qiming Hou, Xin Sun, Kun Zhou, Christian Lauterbach, and Dinesh Manocha*. Memory-Scalable GPU Spatial Hierarchy Construction. *Transactions on Visualization and Computer Graphics*, 17(4):466–474, April 2011.

[Hunt06] *Warren Hunt, Gordon Stoll, and William Mark*. Fast kd-tree Construction with an Adaptive Error-Bounded Heuristic. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 81–88, September 2006.

[Hunt07] *Warren Hunt, William R Mark, and Don Fussell*. Fast and Lazy Build of Acceleration Structures from Scene Hierarchies. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pages 47–54. IEEE, 2007.

[Hunt08] *Warren Hunt*. Corrections to the Surface Area Metric with Respect to Mail-Boxing. In *Proceedings of the 2008 IEEE Symposium on Interactive Ray Tracing*, pages 77–80. IEEE, 2008.

[Intel02] *Intel Corporation*. Intel Pentium III Streaming SIMD Extensions. `http://developer.intel.com/vtune/cbts/simd.htm`, 2002.

[Ize06] *Thiago Ize, Ingo Wald, Chelsea Robertson, and Steven G. Parker*. An Evaluation of Parallel Grid Construction for Ray Tracing Dynamic Scenes. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 47–55. IEEE, September 2006.

[Ize07] *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*, May 2007.

[Ize08] *Thiago Ize, Ingo Wald, and Steven G. Parker*. Ray Tracing with the BSP Tree. In *Proceedings of the 2008 IEEE Symposium on Interactive Ray Tracing*, pages 159–166. IEEE, 2008.

[Ize11] *Thiago Ize and Charles Hansen*. RTSAH Traversal Order for Occlusion Rays. *Computer Graphics Forum*, 30(2):297–305, 2011. (Proceedings of Eurographics).

[Jansen86] *Frederik W. Jansen*. Data Structures for Ray Tracing. In *Proceedings of the workshop on Data structures for Raster Graphics*, pages 57–73, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

[Kalojanov09] *Javor Kalojanov and Philipp Slusallek*. A Parallel Algorithm for Construction of Uniform Grids. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 23–28. ACM, 2009.

[Kalojanov11] *Javor Kalojanov, Markus Billeter, and Philipp Slusallek*. Two-Level Grids for Ray Tracing on GPUs. *Computer Graphics Forum*, 30(2):307–314, 2011. (Proceedings of Eurographics).

[Kammaje07] *Ravi Prakash Kammaje and Benjamin Mora*. A Study of Restricted BSP Trees for Ray Tracing. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007*, pages 55–62, September 2007.

[Karras12] *Tero Karras*. Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics conference on High-Performance Graphics*, pages 33–37. Eurographics Association, 2012.

[Knuth98] *Donald E. Knuth*. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, Second edition, 1998.

[Kopta12] *Daniel Kopta, Thiago Ize, Josef Spjut, Erik Brunvand, Al Davis, and Andrew Kensler*. Fast, Effective BVH Updates for Animated Scenes. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 197–204, 2012.

[Lauterbach06] *Christian Lauterbach, Sung-Eui Yoon, David Tuft, and Dinesh Manocha*. RT-DEFORM Interactive Ray Tracing of Dynamic Scenes using BVHs. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 39–46, September 2006.

[Lauterbach09] *Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha*. Fast BVH Construction on GPUs. *Computer Graphics Forum*, 28(2):375–384, 2009.

[Lee13] *Won-Jong Lee, Youngsam Shin, Jaedon Lee, Jin-Woo Kim, Jae-Ho Nah, Seokyoon Jung, Shihwa Lee, Hyun-Sang Park, and Tack-Don Han*. SGRT: A Mobile GPU Architecture for Real-time Ray Tracing. In *Proceedings of the 5th High-Performance Graphics Conference*, pages 109–119, 2013.

[Lext01] *Jonas Lext and Tomas Akenine-Möller*. Towards Rapid Reconstruction for Animated Ray Tracing. In *Eurographics 2001 – Short Presentations*, pages 311–318, 2001.

[Lloyd82] *Stuart P. Lloyd*. Least Squares Quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[MacDonald89] *J. David MacDonald and Kellogg S. Booth*. Heuristics for Ray Tracing using Space Subdivision. In *Graphics Interface Proceedings 1989*, pages 152–163, Wellesley, MA, USA, June 1989. A.K. Peters, Ltd.

[MacDonald90] *J. David MacDonald and Kellogg S. Booth*. Heuristics for Ray Tracing using Space Subdivision. *Visual Computer*, 6(6):153–65, 1990.

[Magnenat-Thalmann88] *Nadia Magnenat-Thalmann, Richard Laperrière, and Daniel Thalmann*. Joint-Dependent Local Deformations for Hand Animation and Object Grasping. In *Proceedings on Graphics interface '88*, pages 26–33, Toronto, Ont., Canada, Canada, 1988. Canadian Information Processing Society.

[Magnenat-Thalmann91] *Nadia Magnenat-Thalmann and Daniel Thalmann*. Human Body Deformations Using Joint-Dependent Local Operators and Finite-Element Theory. In *Making Them Move: Mechanics, Control, and Animation of Articulated Figures*, pages 243–262. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1991.

[Mora11] *Benjamin Mora*. Naive Ray-Tracing: A Divide-And-Conquer Approach. *ACM Transactions on Graphics*, 30(5):117:1–117:12, October 2011.

[Nah11] *Jae-Ho Nah, Jeong-Soo Park, Chanmin Park, Jin-Woo Kim, Yun-Hye Jung, Woo-Chan Park, and Tack-Don Han*. T&I Engine: Traversal and Intersection Engine for Hardware Accelerated Ray Tracing. *ACM Transactions on Graphics*, 30(6):160:1–160:10, December 2011.

[Ooi87] *B. C. Ooi, R. Sacks-Davis, and K. J. McDonnell*. Spatial k-d-tree: An Indexing Mechanism for Spatial Databases. In *IEEE International Computer Software and Applications Conference (COMPSAC)*, 1987.

[Overbeck07] *Ryan Overbeck, Ravi Ramamoorthi, and William R. Mark*. A Real-time Beam Tracer with Application to Exact Soft Shadows. In *Eurographics Symposium on Rendering*, Jun 2007.

[Pantaleoni10] *Jacopo Pantaleoni and David Luebke*. HLBVH: Hierarchical LBVH Construction for Real-Time Ray Tracing of Dynamic Geometry. In *Proceedings of the Conference on High Performance Graphics*, pages 87–95. Eurographics Association, 2010.

[Parker10] *Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and*

Martin Stich. Optix™: A General Purpose Ray Tracing Engine. *ACM Transactions on Graphics*, 29(4):66:1–66:13, July 2010. (Proceedings of ACM SIGGRAPH).

[Pharr10] *Matt Pharr and Greg Humphreys*. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufman, 2nd edition, August 2010.

[Popov09] *Stefan Popov, Iliyan Georgiev, Rossen Dimov, and Philipp Slusallek*. Object Partitioning Considered Harmful: Space Subdivision for BVHs. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 15–22. ACM, 2009.

[Reshetov05] *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).

[Reshetov07] *Alexander Reshetov*. Faster Ray Packets – Triangle Intersection through Vertex Culling. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pages 105–112. IEEE Computer Society, 2007.

[Rubin80] *Steve M. Rubin and Turner Whitted*. A Three-Dimensional Representation for Fast Rendering of Complex Scenes. *Computer Graphics*, 14(3):110–116, July 1980.

[Santalo02] *Luis Santalo*. *Integral Geometry and Geometric Probability*. Cambridge University Press, 2002.

[Sedgewick98] *Robert Sedgewick*. *Algorithms in C++, Parts 1–4: Fundamentals, Data Structure, Sorting, Searching*. Addison-Wesley, 1998. (3rd Edition).

[Shevtsov07] *Maxim Shevtsov, Alexei Soupikov, and Alexander Kapustin*. Fast and Scalable KD-Tree Construction for Interactively Ray Tracing Dynamic Scenes. *Computer Graphics Forum*, 26(3):395–404, 2007. (Proceedings of Eurographics).

[Shirley03] *Peter Shirley and R. Keith Morley*. *Realistic Ray Tracing*. A K Peters, Second edition, July 2003.

[Stich09] *Martin Stich, Heiko Friedrich, and Andreas Dietrich*. Spatial Splits in Bounding Volume Hierarchies. In *Proceedings of the Conference on High Performance Graphics*, pages 7–13. ACM, 2009.

[Subramanian90] *K. R. Subramanian*. A Search Structure based on kd-Trees for Efficient Ray Tracing. PhD thesis, University of Texas at Austin, December 1990.

[Vinkler12] *Marek Vinkler, Vlastimil Havran, and Jiří Sochor*. Visibility Driven BVH Build up Algorithm for Ray Tracing. *Computers & Graphics*, 36(4):283–296, June 2012.

[Wächter06] *Carsten Wächter and Alexander Keller*. Instant Ray Tracing: The Bounding Interval Hierarchy. In *Rendering Techniques 2006, Proceedings of the Eurographics Symposium on Rendering*, pages 139–149, Aire-la-Ville, Switzerland, June 2006. The Eurographics Association.

[Wald01] *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).

[Wald03] *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)*, pages 77–86, 2003.

[Wald04] *Ingo Wald*. Realtime Ray Tracing and Interactive Global Illumination. PhD thesis, Saarland University, 2004.

[Wald06a] *Ingo Wald and Vlastimil Havran*. On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N). In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 61–70, September 2006.

[Wald06b] *Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G. Parker*. Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM Transactions on Graphics*, 25(3):485–493, 2006. (Proceedings of ACM SIGGRAPH).

[Wald07a] *Ingo Wald*. On fast Construction of SAH-based Bounding Volume Hierarchies. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pages 33–40. IEEE, 2007.

[Wald07b] *Ingo Wald, Solomon Boulos, and Peter Shirley*. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics*, 26(1):6, January 2007.

[Wald08a] *Ingo Wald, Carsten Benthin, and Solomon Boulos*. Getting Rid of Packets: Efficient SIMD Single-Ray Traversal using Multi-branching BVHs. In *Proceedings of the 2008 IEEE Symposium on Interactive Ray Tracing*, pages 49–57. IEEE, 2008.

[Wald08b] *Ingo Wald, Thiago Ize, and Steven G. Parker*. Fast, Parallel, and Asynchronous Construction of BVHs for Ray Tracing Animated Scenes. *Computers & Graphics*, 32(1):3–13, 2008.

[Wald12] *Ingo Wald*. Fast Construction of SAH BVHs on the Intel Many Integrated Core (MIC) Architecture. *Transactions on Visualization and Computer Graphics*, 18(1):47–57, 2012.

[Whitted80] *Turner Whitted*. An Improved Illumination Model for Shaded Display. *CACM*, 23(6):343–349, 1980.

[Woop05] *Sven Woop, Jörg Schmittler, and Philipp Slusallek*. RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005)*, 24(3):434–444, 2005.

[Woop06] *Sven Woop, Gerd Marmitt, and Philipp Slusallek*. B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes. In *Proceedings of Graphics Hardware*, September 2006.

[Wu11] *Zhefeng Wu, Fukai Zhao, and Xinguo Liu*. SAH KD-Tree Construction on GPU. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, pages 71–78. ACM, 2011.

[Yoon06] *Sung-Eui Yoon and Dinesh Manocha*. Cache-Efficient Layouts of Bounding Volume Hierarchies. *Computer Graphics Forum*, 25(3), September 2006. (Proceedings of Eurographics).

[Yoon07] *Sung-Eui Yoon, Sean Curtis, and Dinesh Manocha*. Ray Tracing Dynamic Scenes using Selective Restructuring. In *Rendering Techniques 2007, Proceedings of the Eurographics Symposium on Rendering*, pages 73–84, Aire-la-Ville, Switzerland, June 2007. The Eurographics Association.

[Zhou08] *Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo*. Real-time kd-tree Construction on Graphics Hardware. *ACM Transactions on Graphics*, 27(5):126:1–126:11, 2008. (Proceedings of ACM SIGGRAPH Asia).