State of the Art in Ray Tracing Animated Scenes

Ingo Wald[1,2] William R. Mark[3] Johannes Günther[4] Solomon Boulos[5] Thiago Ize[1,2]
Warren Hunt[3] Steven G. Parker[1] Peter Shirley[5]

1: SCI Institute, University of Utah 2: Intel Corp 3: University of Texas at Austin 4: MPI Informatik, Saarbrücken 5: School of Computing, University of Utah

Abstract

Ray tracing has long been a method of choice for off-line rendering, but traditionally was too slow for interactive use. With faster hardware and algorithmic improvements this has recently changed, and real-time ray tracing is finally within reach. However, real-time capability also opens up new problems that do not exist in an off-line environment. In particular real-time ray tracing offers the opportunity to interactively ray trace moving/animated scene content. This presents a challenge to the data structures that have been developed for ray tracing over the past few decades. Spatial data structures crucial for fast ray tracing must be rebuilt or updated as the scene changes, and this can become a bottleneck for the speed of ray tracing.

This bottleneck has received much recent attention by researchers that has resulted in a multitude of different algorithms, data structures, and strategies for handling animated scenes. The effectiveness of techniques for ray tracing dynamic scenes vary dramatically depending on details such as scene complexity, model structure, type of motion, and the coherency of the rays. Consequently, there is so far no approach that is best in all cases, and determining the best technique for a particular problem can be a challenge. In this STAR, we aim to survey the different approaches to ray tracing animated scenes, discussing their strengths and weaknesses, and their relationship to other approaches. The overall goal is to help the reader choose the best approach depending on the situation, and to expose promising areas where there is potential for algorithmic improvements.

1. Introduction

One of the main elements of a rendering technique is the visibility algorithm. For example, to produce an image it is necessary to determine which surfaces are visible from the eye point, which surfaces are visible from the light(s) and hence not in shadow, and, if global illumination effects are being computed, which surfaces are visible from points on other surfaces. The two most commonly used approaches to the visibility problem are rasterization-based approaches and ray tracing based approaches. The performance of this visibility algorithm is critical for interactive applications.

Rasterization-based approaches are limited to determining visibility for many rays sharing a single origin. They also operate in object order (the outer loop is over objects). These algorithms can be supported very efficiently on special purpose hardware (GPUs), and with hardware and software advancements, GPUs routinely obtain real-time performance for visibility from an eye point or point light even for highly complex models. In addition, they enable a wide array of techniques to produce highly compelling graphics effects at real-time rates. Consequently, virtually all of today's real-time graphics uses GPU-based rasterization, delivering highly compelling imagery at real-time rates.

Ray tracing algorithms [Whi80, CPC84], on the other hand, support arbitrary point-to-point visibility queries and are arguably more powerful for computing advanced lighting effects that require such queries. Off-line rendering has primarily used ray tracing instead of rasterization for these reasons [TL04, CFLB06]. Unfortunately, ray tracing is computationally demanding and has not yet benefited from special purpose hardware, and consequently could not be supported at interactive frame rates until very recently.

With advances in CPU hardware and increased availability of parallel machines, combined with equivalent advances in algorithms and software architectures, ray tracing has reached a stage where it is no longer limited to only off-line rendering. In fact, while the first interactive ray tracers either required large supercomputers [KH95, Muu95, PMS*99] or were limited to small resolutions [WSBW01] and/or simple shading effects [RSH05], there now exists a variety of different interactive ray tracing systems, many of which tackle problems that are not easily possible using a rasterization-based approach [GIK*07, WBS02, BEL*07, PSL*98].

1.1. The need for handling animated scenes

The key to fast ray tracing lies in the use of data structures such as kd-trees, grids, and bounding volume hierarchies that reduce the number of ray-primitive intersections performed per ray. For a long time, ray tracing research concentrated mostly on the effectiveness of these data structures (i.e., how effective each is in reducing the number of primitive operations), and on the efficiency of the traversal and primi-

tive intersection operations (i.e., how fast these operations can be executed on particular hardware). The time for *building* these data structures has typically been ignored – since it is usually insignificant in off-line rendering – and consequently, ray tracing evolved into a field that used data structures and build algorithms that were clearly non-interactive except for trivially simple scenes. Consequently, as ray tracing started to reach interactive rendering performance, it was initially only applicable to walk-throughs of static scenes, or to very limited kinds of animations.

With the advent of ray tracers running at real-time frame rates for certain kinds of static scenes (especially Reshetov's 2005 MLRT paper [RSH05]), it has become clear that build times can no longer be ignored: with up to a hundred million rays per second on a desktop PC, ray tracing has the potential to be used for truly interactive applications like games, but these depend on the ability to perform significant changes to the scene geometry every frame.

In fact, this situation opened up an entirely new research challenge for consideration in ray tracing: to create build algorithms that were fast enough and flexible enough to be used at interactive frame rates. While originally, ray tracing data structures were only considered for their effectiveness and efficiency in rendering, now the build time had to be considered as well. This not only affects which build algorithm is the "best" for any given data structure, but also which data structure to use in the first place. Consequently, many ray tracing data structures are receiving renewed interest even though they had previously been discarded as being too inefficient. In many ways, lessons learned in the early days of batch rendering are being revisited, where the acceleration structure must now pay for itself (with a reduction in rendering time) within the few milliseconds available for a given frame, rather than over several minutes. In fact, the challenge is even greater than in a batch renderer since interactive systems use comparatively few samples per pixel and typically do not have the opportunity to customize the scene for particular viewpoints as is often done in batch rendering.

1.2. Types of animations

One issue that makes it challenging to compare the different approaches to animated ray tracing is that the term "animated" scene is not well-defined, and covers everything from a single moving triangle in an otherwise static scene, to scenes where no two successive frames have anything in common at all. For the remainder of this report, we will use the following terms: A *static* scene is one whose geometry does not change at all from frame to frame[†]; a *partially static* scene is one in which a certain amount of the primitives are moving, while other parts remain static, such as a few characters moving through an otherwise static game environment, or an editing application where small portions of the scene are moving at any given time.

The actual motion of the primitives can either be hierarchical, semi-hierarchical, or incoherent: motion is *hierarchical* if the scene's primitives can be partitioned into groups of primitives such that all of the primitives of a given group are subject to the same linear or rigid-body deformation. Each such group we will call an *object*. The exact opposite of hierarchical motion is *incoherent motion*, where each primitive moves independently of all others; a hybrid situation is *semi-hierarchical* motion, in which the scene can be partitioned into objects whose motion is primarily hierarchical, plus some small amount of incoherent motion within each object (similar to a flock of birds or a school of fish).

In addition to the motion of each primitive, animations can also differ in the way that animation affects the scene topology: Often, an object is stored as a triangle mesh, and animation is performed by moving only the triangle vertices while leaving the connectivity unchanged. We call this special case a *deformable* scene, whereas *arbitrary changes to the scene topology* can also include the change of triangle mesh connectivity, or even the addition or deletion of primitives[‡]. Often, only certain parts of the scene are deformable (e.g., each skinned monster is a deformable mesh), while the scene's overall animation is more complex.

In practice, different applications use different kinds of animation. For example, a design review application is likely to employ either static scenes or semi-hierarchical animation of complete auto or airplane parts; potentially including the addition or removal of complete objects from time to time, but with no non-hierarchical motion at all. A particle simulation, on the other hand, may use completely incoherent motion with frequent addition and removal of primitives, or even completely unrelated primitives every frame. Games, in fact, can employ all kinds of motion at the same time: a flight simulator or first-person shooter, for instance, may contain some static geometry, as well as completely incoherent parts – like explosions. In games there usually are individual objects with mostly hierarchical animation (like airplanes or monsters), but there may be many of them. The motion of the many objects may itself be an example of incoherent motion (e.g. characters appearing and disappearing). In addition, the characters themselves are often skinned, providing a good example of semi-hierarchical motion. It is likely that no one technique will handle all kinds of motions equally well.

2. Problem environment

The problem we target – real-time ray tracing of animated scenes – actually consists of two competing concerns: real-time ray tracing, and handling animated scenes. With the

† Note that we only consider geometric changes – camera, lighting, or shading information do not affect the efficiency data structures, and so will not be considered.

‡ A deformable scene does *not* have to consist of only a single connected object, it can also consist of multiple separate triangle meshes as long as the overall connectivity does not change

goal of interactivity, approaches must be able to generate real-time performance for tracing rays. Ultimately, this requires the use of acceleration data structures such as kd-trees, grids, or bounding volume hierarchies. The relative effectiveness for ray tracing (in particular, for animated scenes) of these acceleration structures varies dramatically depending on many factors, which we discuss below.

This combination of real-time ray tracing with support for animated scenes raises a lot of new issues that we discuss in detail over the course of this report. For static scenes, the choice of a data structure and build algorithm can be determined by looking only at the final rendering performance, since the build cost is not incurred during the interactive session. However, as soon as interactivity for dynamic scenes is attempted, the the time for building or updating the data structure can no longer be ignored.

Exploiting ray coherence for fast ray traversal Much of this report will focus on the methods for building and/or updating ray tracing acceleration structures for animated scenes. However, ray traversal performance is also critical in animated ray tracing systems, especially in systems that trace large numbers of secondary rays. In this section we review an important class of techniques used to provide fast traversal performance in all modern interactive ray tracers.

The rays traced in a typical interactive ray tracer are not organized randomly; there is substantial spatial coherence in the rays that are traced (i.e., they can be grouped together in space). This coherence is particularly strong for eye rays, but it is also present for hard shadow rays, soft shadow rays, and many other kinds of secondary rays. All modern high performance ray tracers exploit this spatial coherence to reduce computation costs. At a high level, there are two strategies for exploiting coherence: beam tracing [HH84] and ray aggregation. Beam tracing performs exact area sampling and thus does not explicitly represent rays at all. On the other hand, ray aggregation explicitly represents rays but amortizes the cost of some of the traversal operations over an entire "packet" of multiple rays.

Most current systems use ray aggregation techniques, which combine several rays into a packet and/or frustum. The first step in that direction was Wald et al.'s "coherent ray tracing" paper [WSBW01], which proposed tracing rays in bundles of four through a kd-tree, and using SIMD to process these rays in parallel; the same concept has since been used in numerous ray tracers, and on a variety of architectures. Using packet tracing allows for amortizing operations including memory accesses, function calls and traversal computations, and permits the use of register SIMD instructions to gain more performance from the CPU. For coherent rays, this can lead to significant performance increases over single ray implementations. Though tracing rays in SIMD, "plain" packet tracing still performs all traversal steps and triangle intersections as a single-ray ray tracer.

An important evolution of packet tracing is the use of frustum- or interval arithmetic-based techniques. Instead of saving only through implicit amortizations and SIMD processing, these techniques go one step further: they use much larger packets than packet tracing, and explicitly avoid traversal steps or primitive intersections based on conservative bounds of the packet of rays. For triangle intersection, this concept was first proposed by Dmitriev et al. [DHS04], who used the bounding frustum to eliminate triangle intersections for cases where the full packet misses the triangle. For traversal, the concept was first proposed by Reshetov et al. [RSH05] who applied it to kd-tree traversal, and used interval arithmetic-based "inverse frustum culling" to cull complete subtrees during traversal. The basic concept was later extended to grids [WIK*06] and BVHs [WBS07], and a large number of modified applications are possible (see, e.g., [BWS06] for a more complete overview). Though more research is required in how such techniques interact with less coherent rays, packet- and frustum techniques are currently the methods of choice for targeting real-time performance.

In a beam tracer, rays are not explicitly represented except perhaps in a final sampling step. Instead, a beam tracer evaluates exact area visibility. Overbeck et al. [ORM07] have recently demonstrated that for scenes composed of moderate to large size triangles, beam tracers are competitive with frustum-based ray tracers for eye rays and soft shadow rays. They achieved this performance through new techniques for using a kd-tree acceleration structure for beam tracing. However, beam tracing performance becomes less competitive for small triangle sizes, since small triangles force a large number of beam splits. An important advantage of beam tracers is that they eliminate the Monte Carlo sampling artifacts produced by traditional ray tracers for soft shadows. Though most of the systems discussed in the remainder of this paper do not use beam tracing, the general discussions on data structures and build/update strategies may apply to a beam tracing approach as well.

Interactivity for animated scenes requires performance both for ray tracing as well as for data structure update/rebuilds. Therefore, we will describe the traversal techniques for kd-trees, BVHs, and grids in Sections 5, 6, and 7, respectively. At the current state-of-the-art, grids, kd-trees, and BVHs all feature fast traversal algorithms; thus, the focus of that paper lies on how to handle dynamically changing geometry. We do so, we fist have to discuss the high-level design issues on how to design a ray tracer for dynamically animated scenes.

3. Overarching tradeoffs

There are a number of candidate approaches for ray tracing dynamic scenes. Before discussing any of these approaches in detail, it is worth considering the general design decisions that must be addressed when attempting interactive ray tracing of dynamic scenes.

Each of these decisions represents one dimension in the overall design space of a ray tracer. We present several dimensions of this design space, discussing the possible

choices and the tradeoffs made with these choices. As some of these decisions are interrelated, we also discuss the effect of one choice on other choices in this space.

Some of these tradeoffs include:

- What kind of acceleration structure should be used? The tradeoffs include using a space partitioning hierarchy vs. an object hierarchy; axis aligned vs. arbitrarily oriented bounding planes; One coordinate system vs. many local coordinate systems; adaptive to geometry vs. non-adaptive; and mechanisms for organizing bounding planes in the data structure.
- How is the acceleration structure built or updated each frame? In large part, this determines the trade-off between build performance and trace performance. In particular, rebuild vs. update; full update/rebuild (entire scene) vs. partial update/rebuild (just portions needed for that frame); fast vs. careful algorithms for choosing bounding planes. These questions are addressed briefly in this section and in more detail in Section 4.
- What is the interface between the application and the ray tracing engine? In particular, how does the application provide geometry to the ray tracing engine: polygon soup vs. sloppy spatial organization (scene graph) vs. ready-to-use acceleration structure? Are there restrictions or optimizations for particular kinds of dynamic movement? Static geometry vs. rigid object movement vs. deformable meshes vs. coherent movement vs. no restrictions? Is geometry represented with just one resolution, or many?

Note that many of these tradeoffs may be substantially different in an interactive system than in a traditional batch ray tracer, where many of these issues are not faced. We discuss several of these tradeoffs below.

3.1. Acceleration structure tradeoffs

As mentioned above, there are a wide variety of acceleration structures that can be used for ray tracing. Specific details of particular acceleration structures will be discussed in Sections 5, 6, and 7, but there are inherent tradeoffs between these techniquesthat we want to contrast in advance.

The choice of an acceleration structure strongly affects the traversal performance, and also can facilitate (or inhibit) the choice of certain algorithms for updating or rebuilding the acceleration structure. Here, we discuss the different kinds of acceleration structures in terms of their fundamental properties. For a more in-depth discussion of spatial data structures in general we refer readers to books by Samet [Sam06, Sam89b, Sam89a] and a survey article by Gaede and Günther [GG98]. For a more in-depth discussion of how ray tracing acceleration structures affect ray tracing traversal performance, we refer readers to the literature listed in the bibliography at the end of Chapter 4 in PBRT [PH04], and in particular to Havran's Ph.D. thesis [Hav01].

Finding the object hit by a ray is fundamentally a search problem, and the data structures used to accelerate that search impose some kind of spatial sorting on the scene. Though a variety of different data structures exist (e.g., grids, kd-trees, octrees, and variant of BVHs), they fall into only two classes: spatial subdivision techniques, and object hierarchies.

3.1.1. Spatial subdivision vs. object hierarchy

Spatial subdivision and object hierarchies are dual in nature: Spatial subdivision techniques uniquely represent each point in space, but each primitive can be referenced from multiple cells; object hierarchy techniques reference each primitive exactly once, but each 3D point can be overlapped by anywhere from zero to several leaf nodes (also see [Hav07]). Grids, octrees, and kd-trees are examples of spatial subdivision, with varying degree of regularity (or "arity" of the subdivision [Hav07]); bounding volume hierarchies and their variants (bounding interval hierarchies, s-kd trees, b-kd trees, ...) are object hierarchies. The advantages and disadvantages of the two approaches follow from these properties.

First, we consider traversal. If we wish to find the first intersection point along a ray, the problem is somewhat simpler for the space partitioning data structures. Each volume of space is represented just once, so the traversal algorithm can traverse these voxels in strict front-to-back order, and can perform an "early exit" as soon as any intersection is found. In contrast, for space overlapping data structures the same spatial location may be covered by different subtrees, and an intersection found in one subtree may later be overwritten by an intersection point in a different subtree that is closer to the origin of the ray (potentially having led to superfluous work). On the other hand, spatial subdivision may lead to visiting the same object several times along the ray, which cannot happen with an object hierarchy; the same is true for empty cells that frequently occur in spatial subdivision, but simply do not exist in object hierarchies.

Space subdivision also generally leads to a finer subdivision (an object hierarchy will never generate cells smaller than the primitive inside), which often encloses objects more tightly. This often leads to fewer primitive intersections, but at the expensive of potentially more work to be performed during building, and possibly more traversal steps. For the same reason the often stated assumption that BVHs consume more memory than kd-trees is not necessarily true: though each node does require more data, the number of nodes, leaves, and triangle references in a BVH is generally much smaller than in a kd-tree; the same observation is true for the cost of traversing the data structure (a more expensive traversal step, but fewer traversal steps).

Second, we consider updates to an acceleration structure as objects move. In a typical object hierarchy data structure, it is easy to update the data structure as an object moves because the object lives in just one node and the bounds for that node can be updated with relatively simple and localized update operations. In contrast, updates to a space partitioning data structure are more complex. If split planes are updated, the changes are not necessarily well localized and may effect other objects.

3.1.2. Axis-aligned vs. arbitrary bounding planes

All of the commonly used acceleration structures rely on planes to partition space or objects. For some types of acceleration structures these planes are restricted, most commonly to be axis-aligned along the *x*, *y*, or *z* axis. However, it is also possible to allow arbitrarily oriented bounding planes, as is done in a general binary space partitioning (BSP) tree. The advantages of using axis-aligned planes include: (a) The plane's orientation can be represented with just two bits, rather than two or more floating point numbers, (b) Intersection tests are simpler and faster for axis-aligned planes, (c) Numerical precision and robustness issues in ray-plane intersection are easier to characterize and solve for axis-aligned planes, (d) Using only axis-aligned planes significantly reduces the dimensionality of the search space for building efficient data structures, and building efficient good axis-aligned data structures is well understood. Conversely, the advantages of using arbitrarily-oriented planes include: (a) Arbitrarily aligned planes can bound geometry more tightly than axis-aligned planes; and (b) Some strategies for incremental update of an acceleration structure might benefit from the ability to arbitrarily adjust the orientation of bounding planes to accommodate rotations of objects.

There has been very little investigation to date of general BSP trees or BVHs with non-axis aligned bounding primitives as ray tracing acceleration structures, even though both have been used for collision detection (see e.g. [LCF05, GLM96, HEV*04]).

3.1.3. Adapt to geometry vs. non-adaptive

For spatial subdivision techniques, one more option is the mechanism for subdividing space. In some acceleration structures the location of subdivision planes is chosen so as to adapt to the geometry in the scene (e.g. a kd-tree), whereas in other acceleration structures the locations of bounding planes are predetermined, without looking at the geometry in the scene (e.g., a grid or octree). In this second case, some acceleration structures are still able to partially adapt to the scene geometry by adjusting their topology (e.g. an octree or grid with variable depth), whereas other acceleration structures do not adapt at all to scene geometry (e.g., a regular, non-hierarchical grid).

The advantage of the most highly adaptive data structures is that they are able to compactly and efficiently represent almost any scene, including those with highly variable density of geometry such as the "teapot in a stadium". For this same reason, they also provide good traversal performance on virtually any scene (also see [Hav01]).

When rays are traced in aggregates such as packets, frusta, or beams – which today is widely believed to be a prerequisite to reaching high performance – there can be dramatic differences in the traversal performance characteristics of different acceleration structures. For grids, only one packet-based traversal scheme is known today [WIK*06], and since it is based on frustum traversal, it requires more than 4 rays to benefit from the frustum traversal; with 4 or less rays the performance advantages over a single ray grid are much lower and it can even perform worse, or fall back to single-ray traversal. In addition, the grid requires highly coherent rays to perform efficiently. Adaptive data structures, on the other hand, seem to be more friendly to different packet configurations. In part, this is because the hierarchical nature of adaptive data structures allows much of the traversal work to be done at coarse spatial scales, where the amortization of costs for large packets is especially effective, even when the packets are less coherent. It is possible that a hierarchical grid or an octree might be similarly effective for large packets with less coherence, but this question has not been studied in detail yet.

Adaptivity also affects the construction of the acceleration structure. In general, adaptive data structures are more expensive to construct than non-adaptive data structures. There are several reasons for this. First, adaptivity fundamentally requires that more decisions be made, and these decisions require computation. Second, when inserting a new object into an adaptive data structure, some traversal of the data structure is required whereas none is required for a non-adaptive data structure such as a grid. Finally, parallelization of acceleration structure construction is more complex for adaptive data structures than for non-adaptive data structures. From an algorithmic standpoint, building an adaptive data structure is related to sorting, and typically requires super-linear time, while building a regular data structure is very similar to triangle rasterization, and can be done in an single pass.

3.1.4. Build time vs. build quality

For every data structure, there are different ways of building that data structure for any given scene; commonly, there is a trade-off between build quality (i.e, how good the data structure is at minimizing render cost) and build time. For example, a kd-tree can be built over bounding boxes or over actual triangles (involving lots of costly clipping operations) and this difference can have an impact on render performance of 25% and more [Hav01, WH06].

When building or updating a hierarchical acceleration structure whose bounding planes adapt to geometry (i.e. nearly all acceleration structures except grids), the build/update algorithm must decide how to organize geometry into a hierarchy and choose locations for bounding planes. Heuristics for evaluating the cost of any given tree configuration exist (we will go into more detail below), but with an exponential number of possible tree configurations, finding the globally best configuration is computationally intractable. The best known heuristic is the greedy Surface Area Heuristic (SAH) [GS87], which is explained in somewhat more detail below, as well as in Havran's thesis [Hav01], and in Pharr and Humphreys' book [PH04] (along with example code). However, though the greedy SAH algorithm has the same asymptotic complexity as a spatial median split (and in practice also exhibits near linear cost [WH06]), evaluating lots of potential split planes incurs a significant cost.

At the other extreme, very simple heuristics can be used such as placing a split plane at the median point within the current node. However, the acceleration structures produced by these algorithms generally exhibit significantly poorer traversal performance than those built by the greedy SAH especially when the density of geometry varies significantly within the scene.

Recently, algorithms have been developed that are designed to approximate the greedy SAH (we discuss these algorithms in more details below, see Section 5.1). For a moderate impact on build quality, these usually are substantially faster to build, and typically offer interactive rebuilds [HSM06, PGSS06].

3.2. System Architecture tradeoffs

As ray tracing becomes practical for real-time applications it becomes increasingly important to consider how the core ray tracing engine, data structures, and algorithms should interact with the data structures and code of a real application. For example, how might a virtual reality system or game application use a ray tracing-based rendering engine?

There are currently two broad schools of thought on this question. The first, embodied in the OpenRT interface [WBS02], argues that the interface between the rendering engine and application should be as similar as possible to the immediate mode APIs used in Z buffer systems such as OpenGL (and thus, ease the transition to ray tracing). The second, originally advocated by Mark and Fussell [MF05] and implemented in Razor [DHW*07] argues that it is necessary to thoroughly reconsider this interface for ray tracing systems and adopt an approach that more tightly couples the application's scene graph data structure to the ray tracing rendering engine.

3.2.1. Polygon soup vs. scene graph vs. ready-to-use acceleration structure.

Hierarchical motion and most kinds of deformable motion can be readily expressed via hierarchical data structures such as a scene graph passed between the application and the rendering engine (and most applications actually do this). Fully incoherent motion must be expressed essentially in the same way as it is in a Z-buffer system: by passing unorganized polygons as a "polygon soup" from the application to the rendering engine. However, incoherent motion is often spatially localized, in which case a hierarchical data structure such as a scene graph can at least isolate these scene parts from other, more hierarchically organized scene parts.

It is also possible for the application to pass a completely built acceleration structure to the rendering engine. This approach is appropriate either for static geometry; or for an acceleration structure that can be incrementally updated; or for a "low quality" acceleration structure such as a scene graph that will only be used by the rendering engine to build a higher quality acceleration structure.

In choosing the data structures passed between the application and the rendering engine there is a tension between the needs of the application and the needs of the rendering engine. A polygon soup or scene graph is often most natural for the application, while an acceleration structure is most natural for the rendering engine. The considerations involved are complex and to some extent will depend on the particular kind of application.

3.2.2. Pre-transformed geometry

Traditional acceleration structures typically use a single global coordinate system to represent all objects and bounding planes in the acceleration structure. It is also possible to use hierarchical transformations to enable different coordinate systems in different portions of the acceleration structure [LAM01, WBS03]. Typically this is done by including coordinate-transform nodes in the acceleration structure.

There are several advantages to supporting local coordinate systems in the acceleration structure. First, as will be discussed in Section 4.3, this mechanism provides an extremely simple way to animate entire objects by translation or rotation [LAM01, WBS03] – only the coordinate-transform node needs to be changed. For this same reason, coordinate-transform nodes are almost always supported in scene graph data structures [Ebe05]. Second, in an acceleration structure such as a kd-tree that restricts bounding planes to be axis-aligned in the local coordinate system, the coordinate-transform node provides a mechanism for effectively allowing arbitrary orientation of the planes (also see [GFW*06]). On the downside, systems that use local coordinate systems for animation are limited to supporting hierarchical animation of rigid bodies; and the data structure and traversal algorithm become more complex, potentially slowing down the traversal.

3.2.3. Level-of-detail and multiresolution surfaces

Support for ray tracing of multiresolution surfaces and/or objects can have a pervasive effect on the entire ray tracing system. A full discussion of these issues is beyond the scope of this paper but we will highlight some of the key interactions between multiresolution surfaces and support for dynamic geometry.

First, we note that there are two basic approaches to supporting multiresolution surface patches. The first is to tessellate the surface patch into triangles at the desired resolution (e.g. [DHW*07]), and the second is to directly intersect rays with the surface patch (e.g. [BWS04, AGM06]). Both of these techniques have been used for many years in off-line rendering systems.

In a system that tessellates the surface patch into triangles, all surfaces in the system effectively become dynamic geometry in the sense that the tessellation can change from frame to frame. This situation provides an especially strong incentive to efficiently support dynamic geometry.

There are two different forms of multiresolution ray tracing. The first, *eye point directed multiresolution*, sets the tessellation rate of a surface patch-based on its distance from the eye point. This approach is similar to what is done in

Z buffer systems, and insures that there is only one representation of a surface patch in any particular frame. It is relatively straightforward to implement. The second, *ray directed multiresolution*, allows each ray to independently set the resolution of the surfaces it intersects, using information taken from ray differentials [Ige99, SW01]. In this approach a surface patch may have several different representations, each of which is used by a different ray. In a system that tessellates patches into triangles, this second category of multiresolution requires substantial changes to the acceleration structure, traversal algorithms, and intersection algorithms to perform efficiently and avoid cracking artifacts [DHW*07]. Hybrids of these two approaches are also possible, as is used by Pixar's PhotoRealistic RenderMan batch renderer [CLF*03, CFLB06].

In a system that supports multiresolution surfaces – either via tessellation of surface patches or via other mechanisms such as progressive meshes – this capability must be exposed to the application. For example, Razor accepts Catmull-Clark subdivision patches [DHW*07], and Benthin's free-from ray tracing system accepts cubic Bezier splines and Loop subdivision surfaces [BWS04]. However, the impact of multiresolution surfaces is not limited to the application interface; they can dramatically affect almost all aspects of the ray tracing engine as will be discussed below.

There are several advantages to multiresolution surfaces. The first is the ability to represent curved surfaces without requiring a high a-priori tessellation rate that would result in an extremely high polygon count. The second is the ability to represent the scene database more compactly than would be possible with a polygon representation. In an appropriately designed system this more compact representation can reduce memory bandwidth requirements. Multiresolution surfaces also offer a form of anti-aliasing for highly detailed objects.

The disadvantage to multiresolution surfaces is that they add considerable complexity to the system. They may also reduce performance due to the need for on-demand tessellation and/or complex ray-surface intersection algorithms.

4. General Acceleration Structure Strategies

Given the design space of acceleration structures presented above, we now discuss the tradeoffs that are common with all of the known acceleration structure strategies.

Arguably the most important question that arises in designing an interactive ray tracing system for dynamic scenes is how to rebuild or update the acceleration structure each frame (or time step). We note that this same general problem has been studied extensively in the context of collision detection [JP04, LAM05, vdB97, TKH*05, LM04], although the goals and constraints are somewhat different in ray tracing. The problem is also a specialized form of the sorting problem so Knuth's book on this topic [Knu98] can often provide valuable insights.

In general, there is a trade-off between build time and traversal performance: investing more time into building to get a better data structure can save time spent in traversal and intersection, but is only worthwhile if the savings in traversal are not outweighed by the additional time spent on preparing the data structure. The costs of build (or update) and of traversal are in turn strongly affected by two broad factors: the characteristics of the rendering task, and the choice of acceleration structure. We'll discuss the key characteristics of the rendering task first, since those in turn strongly interact with the choice of acceleration structure.

The most important characteristics of the rendering task that affect the trade-off between build time and traversal performance are:

- **Kinds of motion in the scene**. As discussed earlier in Section 1.2, there are a variety of kinds of motion. For example, gradual movement of rigid objects presents a very different kind of problem than random creation and deletion of geometry every frame. This first case can be handled easily with minor updates to an acceleration structure while the second case strongly favors approaches that can efficiently rebuild the entire acceleration structure. Techniques and systems may or may not take advantage of the properties of that motion. Separating static objects from dynamic objects is commonly employed. For example, a static building or tree might have its own static acceleration structure built at great expense, while a dynamic character might benefit from different strategies. Characters, vehicles, and other common objects usually experience limited deformations and seldom change topology. These properties can be exploited in the acceleration structure.
- **Geometric complexity of the scene**. More specifically, the total amount of geometry, the amount of geometry that is visible to rays, and the variation in spatial density of the geometry are all important to the choice of algorithms.
- **Total number of rays**. All other things being equal, if more rays are being traced it may be worthwhile to invest more time to build a good acceleration structure. Ray count is determined primarily by image resolution, sampling density, and by the average number of secondary rays per primary ray.
- **Kind of rays**. Secondary rays, especially those for area light sources and hemisphere sampling, may access the acceleration structure less coherently than primary rays.
- **Ray aggregation strategy**. Traversal time can be strongly affected by the choice of ray aggregation strategy (e.g. frusta, packets, beams, etc). For example, a frustum tracer may benefit less from tighter fitting leaf nodes than a single-ray tracer.

Build (or update) time and traversal performance are also strongly affected by the choice of acceleration structure. However, the strengths and weaknesses of each acceleration structure should be evaluated in the context of the characteristics of the rendering task just discussed. Generally speaking, grids are currently considered to be fastest to build but the least efficient for traversal/intersection; kd-trees to be the most efficient ones for traversal/intersection but most costly

ones to build, and BVHs somewhere in-between in build time, and close to kd-trees for traversal.

These considerations are weighted differently for interactive ray tracers than they have been for other kinds of systems in the past. In an offline rendering system, any build strategy that is sub-quadratic (lower than $O(N^2)$ for N primitives) in time and linear ($O(N)$) in memory is considered practical. With a large number of rays traced, these acceleration structures easily recouped the cost of build within a single frame, or the acceleration structure could be computed offline and stored with the model. In an interactive system with static geometry, the cost of building an acceleration structure can be amortized over many frames or can be computed as a preprocess. Thus, build performance is even less of a concern.

In the following subsections, we present the design space for systems that support scenes whose geometry varies from frame to frame. Most of these strategies can be used with a variety of different acceleration structures.

4.1. Rebuild vs. update vs. static

There are three fundamental approaches to obtaining an acceleration structure for dynamic geometry. The first is to rebuild the acceleration structure from scratch. The second is to update the acceleration structure, typically starting from the one used in the previous frame. The third is to recognize that in some cases part or all of the scene can be treated as static geometry (possibly with respect to some moving local coordinate system), eliminating the need to modify part of the acceleration structure.

The most general method for handling dynamic scenes is to rebuild the data structure from scratch every frame. The primary advantages to this approach are that it can handle arbitrary movement of dynamic geometry, it is simple, and it is relatively straightforward to ensure that the acceleration structure is well optimized. The primary disadvantage to this approach is that it can be expensive to rebuild the acceleration structure from scratch especially for large scenes and hierarchical, adaptive data structures.

In practical applications there is significant coherence among successive frames. Instead of rebuilding each frame's acceleration structure from scratch, it is possible to update the bounding planes and possibly some of the topology of previous frame's acceleration structure to be correct for the new frame's configuration of geometry. The feasibility of this approach depends significantly on the actual acceleration structure and on the kind of motion present in the scene. The primary advantage to this approach is that it is often less time consuming to update the data structure than to rebuild it, because much of the information about the geometry sort can be reused. This approach also has the potential to facilitate the use of other forms of frame-to-frame coherence (for example, in light transport). The primary disadvantage of this approach is the tendency for the quality of the acceleration structure to degrade after some number of updates, especially if the topology of the acceleration structure is not updated. There are strategies to mitigate this problem (discussed later in the paper), but they introduce additional complexity. To date, incremental updates have been proposed primarily for BVHs [LYTM06, WBS07, YCM07, EGMM06, IWP07], and to a lesser degree also for grids [RSH00] and kd-trees [GFW*06].

Finally, if large parts of the scene are static, build/update time for those parts of the scene can be eliminated by storing the static geometry in a separate precomputed acceleration structure or sub-structure. This approach is very fast, but obviously is limited to geometry that isn't fully dynamic. This strategy is usually hybridized with more general strategies. For example, a system that animates rigid objects using a hierarchy of transformations can precompute the acceleration structure for each object and refer to each of these precomputed acceleration structures from within a rebuilt/updated top-level acceleration structure via a coordinate-system rotation node [LAM01, WBS03]. This strategy is discussed in more detail in Section 4.3.

Various hybrids of all three of these general strategies are possible. In particular, when rebuilding or updating an acceleration structure it may make sense to avoid touching those parts of the acceleration structure that are known not to have changed (because their geometry is not currently moving), and reuse those parts of the acceleration structure from the previous frame.

4.2. Complete vs. partial update/rebuild

The simplest approach to updating or rebuilding the acceleration structure is to recreate the *entire* acceleration structure, or even just the objects that moved. We refer to this strategy as the *complete rebuild/update* strategy. This approach has the advantage of being simple and robust, but it also can be very expensive for scenes with large amounts of geometry.

There is an opportunity to reduce this cost by realizing that a full build/update does unnecessary work. When rendering any particular frame of a scene with high depth complexity, most of the geometry is occluded. This property is particularly true for primary rays, but also applies to secondary rays. There is no need to include this occluded geometry in the acceleration structure. Thus, if we can determine which geometry is needed, it is only necessary to *partially* rebuild or update the acceleration structure, so that it just includes the visible and perhaps nearly-visible geometry.

There is one difficulty with this approach: to determine which geometry should be placed into the acceleration structure we need to know which geometry is visible. But that in turn looks a lot like the original ray tracing problem we were trying to solve. For this technique to work in practice, the entire scene geometry must already reside in a hierarchical spatial data structure, i.e. an acceleration structure of sorts. It is perfectly acceptable for this initial acceleration structure to be a low quality one such as the bounding volume hierarchy associated with a typical scene graph. If we

traced rays through this data structure directly, traversal performance would be terrible, but the data structure does provide sufficient spatial sorting to allow the system to determine which geometry must be used to build or update the high quality partial acceleration structure. With this scheme, the partial high-quality acceleration structure is built on demand (i.e. lazily) from the complete low-quality acceleration structure.

There are several advantages to this technique. First, it allows the system to rebuild or update just the needed parts of the high-quality acceleration structure. This reduces build time and reduces storage required for the high-quality acceleration structure. Furthermore, the update/rebuild work that still needs to be done is more efficient, because the hierarchy information from the low-quality acceleration structure provides a partial presort of the geometry. A second advantage of this technique is that it allows the low-quality and high-quality accelerations structures to be more highly tuned for their particular use than in a system that uses just a single acceleration structure, avoiding the tension between scene management and ray tracing that occurs in a system organized around a single acceleration structure. Finally, there is an opportunity to perform other useful work when copying geometry from the low-quality acceleration structure to the high-quality acceleration structure. For example, this is an opportune point at which to tessellate curved surfaces into triangles.

There are also several disadvantages to this technique. First, it requires that the rendering engine be more tightly coupled to the software layer above it than has traditionally been the case, and that this layer above maintain the low-quality acceleration structure (i.e. scene graph). Second, this technique is substantially more complex than non-lazy techniques. In particular, lazy construction of the acceleration structure may be more challenging to parallelize, and may be more challenging to support on specialized hardware. Third, when depth complexity is low this technique does not provide any significant performance advantage.

Most interactive ray tracing systems use complete rebuild/update, but the Razor ray tracing system [DHW*07] (Section 5.2) uses partial build of a high-quality kd-tree from a low-quality BVH scene graph. Similar techniques could be also used to build a high-quality BVH acceleration structure from a scene graph, but no results with interactive performance have been published for this approach yet.

4.3. Hierarchical animation through multi-level hierarchies

As briefly mentioned earlier in Section 4.1, if motion in a scene is organized hierarchically, the acceleration structure update can also be handled in a hierarchical way. If an entire group of primitives is subject to the same linear transformation, one can also transform the ray with the inverse transformation, and intersect those transformed rays with the original geometry. As the geometry itself is never actually transformed, the object's own acceleration structure data structure remains valid, and can thus be pre-built off-line. If more than one of these linearly transformed objects exist, one can build an additional acceleration structure over the world-space instances of these objects, and transform the rays to the respective objects' local coordinate systems during traversal (an instance is a reference to a static object, plus a transformation matrix).

The core idea of using multiple hierarchy levels and transforming rays instead of objects was first proposed by Lext at al [LAM01]; in an interactive context it was first applied by Wald et al. with the OpenRT system [WBS03]. In their paper, the authors also proposed to use a two-level hierarchy for mixing different strategies; in their case by encapsulating incoherent motion into special objects that are then rebuilt per frame. As a side effect, this approach also supports instancing since the same object can be referenced by multiple instances. In addition, it can also be used to support pose-based animation. In many games, animating an object is done by having multiple predefined poses for each character, and switching those per frame depending on what pose is required. Using the multi-level approach, this can be handled by having one separate object for each pose, and only instantiating the one required per frame. This was, for example, used in the proof-of-concept "Oasis" and "Quake4/RT" games [SDP*04, FGD*06], and has proven very effective.

Essentially, the core idea of this approach is to trade off per-frame rebuilding against per-frame ray transformations during traversal. This reduces the complexity of per-frame rebuilds to only rebuilding the top-level hierarchy over the instances. The obvious disadvantages of this approach are a) that performance may degrade if objects overlap significantly, b) that transforming rays adds an additional cost to traversal, and c) that the information on what primitives to group into which objects has to be supplied externally. On the positive side, the concept is very general, and, in particular, completely orthogonal to all other techniques discussed below. It is also the *only* currently proposed technique that remains sub-linear in the number of (visible) triangles, as all other techniques have to at least update all of the triangle positions.

4.4. General strategies for speeding up rebuild

A straightforward rebuild algorithm (discussed originally in Section 4.1) can be made faster in one of two ways: improving single-thread build performance or parallelizing the build. Higher single-thread build performance obviously requires low-level optimizations like the use of SIMD extensions [PGSS06, HSM06, SSK07], but primarily revolves around investigating build algorithms with a quality-vs-speed trade-off, i.e., in simpler build strategies that yield data structures with inferior traversal performance, but produce them at much faster rates [PGSS06, HSM06, HHS06, WK06].

Parallel rebuilding for real-time builds has only been considered fairly recently; parallel data structure builds have been studied in the context of static scenes (e.g., [BGES98,

Ben06]), but with the growing availability and size of multi-core systems is currently receiving increased attention. Fast, parallel, and scalable builds today are available for both grids [IWRP06] and kd-trees [SSK07], both of which are discussed in more detail below. The Razor system parallelizes its kd-tree build by allowing each processor to lazily build its own kd-tree [DHW*07]. While this strategy is actually building multiple kd-trees in parallel, each on its own core, each tree is incomplete, so when ray tracing work is carefully allocated to processors this strategy does surprisingly little redundant work and has been demonstrated to be effective for eight cores. Parallel BVH building has received less attention, but the parallel kd-tree builds should also generalize to BVHs.

4.5. Fast rebuild with application/scenegraph support

Initial work on ray tracing of dynamic scenes assumed that it would be necessary to restrict the kind of motion or to use a poor-quality (but quick to build) acceleration structure in order to achieve interactive frame rates. However, recent work [DHW*07] has shown that by exploiting knowledge of the scenegraph and application, combined with a lazy rebuild strategy, one can achieve efficient acceleration structures for arbitrary dynamic motion. This approach will be discussed in more detail for kd-trees in 5.2.

5. Kd-tree-based approaches

In this section we present a variety of approaches for ray tracing dynamic scenes using a kd-tree acceleration structure. Some of these techniques may be applicable to other acceleration structures, but are discussed here because they were first developed for use with kd-trees.

Kd-trees are considered by many to be the best known method to accelerate ray tracing of static scenes. Their primary disadvantage for dynamic scenes is that updates are very costly (at least efficient update methods are not yet known). Various approaches have been developed to avoid this limitation. The first approach is to accelerate the construction of the tree by clever approximations and by parallelization. The goal is to build the kd-tree fast enough from scratch every frame, thus supporting arbitrary modification of the scene geometry. The second approach is to avoid the rebuild of the kd-tree by transforming rays instead of geometry. And finally, a kd-tree can instead be lazily rebuilt, restricting the construction to subtrees that are actually traversed by rays.

5.1. Fast kd-tree construction using scanning/streaming

Several algorithms have been developed to (re-)construct kd-trees from quickly [PGSS06, HSM06, SSK07]. These algorithms have very few assumptions on the input data: they all work with a "triangle soup" (or more precisely "primitive soup") and thus retain maximal flexibility regarding dynamic changes of the scene. If additional information from a scene graph is exploited, the constructions can be accelerated further (described further in Section 5.2).

A kd-tree construction is typically based on a comparison-based sorting operation; thus, it cannot be expected to find faster algorithms than $O(N \log N)$. Therefore, a fast kd-tree builder must concentrate on lowering the constants rather than asymptotically faster algorithms. Construction speed can be improved by algorithms that are better adapted to current hardware architectures, by applying parallelization techniques to exploit the power of multiple CPU cores, and by introducing approximations to save computation. In particular, the currently best know heuristic to produce the highest-quality kd-tree for ray tracing, the *surface area heuristic* (SAH) [MB89], can be approximated without significant degradation of kd-tree quality.

Before discussing the details of the fast construction algorithms we will first give some background regarding the surface area heuristic and briefly cover previous construction methods.

5.1.1. The surface area heuristic

The surface area heuristic [GS87, MB89, MB90, Sub90] provides a method for estimating the cost of a kd-tree for ray tracing based on assumptions about the distribution of rays in a scene. Minimizing this expected cost during construction of a kd-tree results in an approximately optimal kd-tree that provides superior ray tracing performance in practice.

Ray tracing costs are modeled by the SAH by assuming an infinite uniformly distributed rays, in which the probability P_{hit} of a ray hitting a (convex) volume V is proportional to the surface aread SA of that volume [San02]. In particular, if a ray is known to hit a volume V_S, the probability of hitting a sub-volume V_S is

$$P_{hit}(V|V_S) = \frac{SA(V)}{SA(V_S)} \tag{1}$$

The expected cost C_R for a random ray R to intersect a kd-tree node N is given by the cost of one traversal step K_T, plus the sum of expected intersection costs of its two children, weighted by the probability of hitting them. The intersection cost of a child is locally approximated to be the number of triangles contained in it times the cost K_I to intersect one triangle. We name the children nodes N_l and N_r and the number of contained triangles n_l and n_r, respectively. Thus, the expected cost C_R is

$$\begin{aligned} C_R &= K_T + K_I \left[n_l P_{hit}(N_l|N) + n_r P_{hit}(N_r|N) \right] \\ &\overset{(1)}{=} K_T + \frac{K_I}{SA(N)} \left[n_l SA(N_l) + n_r SA(N_r) \right] \end{aligned} \tag{2}$$

During the recursive construction of the kd-tree one needs to determine where to split a given kd-tree node into two children or to create a leaf. According to the SAH, the best position (and dimension) is when C_R is minimal. If the minimal C_R is greater than the cost of not splitting at all ($K_I \cdot n$) a leaf is created. The minimum of C_R 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_R 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.

5.1.2. Brief discussion of previous construction methods

Consequently, previous methods to efficiently construct SAH kd-trees concentrated on the fast evaluation of the primitive counts. One method is to sort the primitives (and thus the split candidates) at one dimension. Then one iterates through the split candidates and incrementally updates the primitive count of the children. The overall complexity of this construction method is $O(N \log^2 N)$ because sorting is needed for every split of a node. By sorting the primitives once in advance–and maintaining the sort order in each splitting stage–this complexity can be reduced to $O(N \log N)$, the theoretical lower bound [WH06].

Although reaching the optimal asymptotic complexity of $O(N \log N)$ this construction algorithm has several main drawbacks: First, the sort at the beginning makes lazy builds (Section 5.2) ineffective, because this sorting step requires $O(N \log N)$ operations. Second, maintaining the sort order during splitting introduces random memory accesses that severely slows down construction speed on cache-based hardware architectures.

Realizing these problems, new state-of-the-art construction methods avoid sorting, and emphasize streaming/coherent memory access. This is possible by giving up exact SAH evaluation. Instead, the SAH function is subsampled and its minimum is only approximated.

The approaches presented in [HSM06] and [PGSS06] both use a sampling and reconstruction approach to finding the global minima in the SAH function. The former uses two sets of 8 samples each and uses a linear approximation for the distribution of the geometry in the scene and a quadratic approximation to the SAH function. The later uses 1024 samples and uses a linear approximation of the SAH function. Both have been shown to produce relatively high quality trees for ray tracing, with (trace) performance similar to a full SAH build trees. These scanning/binning approaches have the advantage of simple, system friendly implementations and $O(N \log N)$ asymptotic performance.

5.1.3. SIMD scanning

An approach to improve memory performance on a modern architecture while approximating the SAH was provided by Hunt et al. [HSM06]. This method uses two passes to adaptively sample the distribution of geometry in the scene and then constructs an approximate SAH cost function based on these samples. In the first pass they take eight uniformly distributed samples, which can be performed very efficiently in one scane over the primitives by using several SIMD registers to count the primitives on either side of eight planes. In the second pass the gathered information is used to adaptively place eight additional samples in a way that minimizes the overall error of the approximation of the SAH cost function.

The end result of these two sampling passes is a piecewise quadratic approximation to the SAH cost function. Hunt et al. also proved that the error of this approximation scheme is bounded by $O(1/k^2)$ with k being the number of samples per pass. To choose the split plane location for the node, they consider the *analytic* local minima of each of the $2k-1$ piecewise quadratic segments, and place the split plane at the overall minimum.

When the recursive construction reaches the lower levels of the kd-tree with only few primitives left the piecewise quadratic approximation becomes inexact. Therefore Hunt et al. switch to the exact SAH cost function evaluation once the number of primitives in a node to split is below 36. The exact evaluation is done using the same SIMD scanning algorithm, but this time placing the samples at all split candidates. Although theoretically a slow $O(N^2)$ algorithm this approach turns out to actually be faster than all other known strategies (including the sorting method described above), due to a machine-friendly implementation.

5.1.4. Streamed binning

Popov et al. [PGSS06] linearly approximate the SAH cost function with 1024 uniformly distributed samples. Using a binning algorithm, they efficiently evaluated the SAH cost function at these sample locations in two phases: In a first phase they stream over the primitives and bin the minimum and the maximum extent (the bounds) of each primitive. The second phase iterates over the bins and reconstructs the counts of primitives to the left and to the right of the border of each bin (these are the sample locations) with partial sums. When performing the split of a node, it's children are binned at the same time, which considerably reduces memory bandwidth.

The authors additionally present a conservative cost function sampling method. Exploiting certain properties of the SAH cost function they can identify and prune bins that cannot contain the minimum of the cost function. The remaining bins are then resampled with 1024 additional samples. With very few iterations only one bin with only one primitive border remains, quickly yielding the *true* minimum and thus the *optimal* split position.

Because binning becomes inefficient when the number of primitives is close to the the number of bins Popov et al. also revert to exactly evaluating the SAH cost function. Once the working set fits into the L2 cache they switch to an improved variant of the classical $O(N \log N)$ SAH algorithm (Section 5.1.2). They use a fast $O(N)$ radix sort [Knu98, Sed98] instead of an comparison-based $O(N \log N)$ sorting algorithm to sort the primitives of this subtree. Note that radix sort is only efficient when the working set fits in the cache because it accesses memory in random order. Additionally the random memory accesses for maintaining the sort order during splitting – originally the motivation for the streamed

binning algorithm – stay now in the L2 cache and are thus not a performance problem any more.

5.1.5. Results and discussion

Both of these scanning and binning algorithms are an order of magnitude faster then previous work with a construction performance of about 300k and about 150k primitives per second in [HSM06] and [PGSS06], respectively. This speed allows for rebuilding the kd-tree per frame to handle arbitrary dynamic changes in smaller to medium sized scenes. Hunt et al. also report numbers for a variant where they trade kd-tree quality for construction speed: When scanning only one dimension per node instead of all three they can construct a kd-tree for the 1.5M triangle Soda Hall scene in only 1.6 s.

Although approximating the SAH cost function the resulting kd-trees are still of high quality, coming close to an "optimal" SAH kd-tree within 2%–4% (measured in expected ray tracing cost and cross-checked with actual rendering speed). This also means that the conservative sampling approach using rebinning of Popov et al. will only be necessary if the optimal kd-tree quality is desired.

Comparing the timings [HSM06] is significantly faster than [PGSS06]. Although different hardware was used for both papers, the reason is most likely due to the simpler (and thus faster) inner loop of [HSM06]. Additionally, Popov et al. switch much earlier to the more exact but slower constructing algorithm than Hunt et al.

Both sampling algorithms have different properties and asymptotic behaviors. With n being the number of primitives in the current node and k being the samples to take SIMD scanning has complexity $O(n \cdot k)$ while streamed binning has complexity $O(n+k)$. This is the reason that Hunt et al. can only afford a small number of samples (16), whereas Popov et al. can easily handle a large number of samples (1024).

The small number of samples of [HSM06] is redeemed by the adaptive placement of the samples, and a quadratic approximation of the cost function. In [PGSS06] the uniform placement of many samples results in a linear approximation of the cost function. However, because we are only interested in the minimum of the SAH cost function it is not necessary to minimize the total approximation error as done in [HSM06]. Additionally, the cost function is smoother with larger n suggesting that the quadratic approximation with few samples will be quite accurate with larger n whereas the linear approximation with many samples will be more accurate with smaller n. Thus it could be advantageous to combine the ideas of both papers when implementing a fast kd-tree builder.

5.1.6. Scalable parallel kd-tree construction

While these methods increase single-thread performance of kd-tree construction, further performance improvements can be achieved through multi-thread parallelization. This is becoming increasingly important with the ongoing trend in CPU and GPU architectures to deliver increased multi-core parallelism.

Figure 1: *This dynamic scene with 63k static and 171k animated triangles can be ray traced with 7.2 fps at 1024^2 pixels on a four core 3 GHz Intel Core 2 Duo including shadows from two point lights [SSK07].*

Until recently, the parallel, multi-threaded construction of kd-trees has not been the focus of research; Benthin [Ben06] showed parallelized kd-tree construction, but with only two threads. A preliminary attempt at parallel construction was also shown in [PGSS06], but with rather limited success due to poor scalability. The difficulties in multi-threaded kd-tree (or any other acceleration structure) construction lie in balancing the work, and in avoiding communication and synchronization overhead.

The first successful, scalable parallelization of kd-tree construction was shown by Shevtsov et al. [SSK07]. Their kd-tree builder closely follows Popov et al.'s streamed binning [PGSS06]. Each thread independently builds its own subtree in dedicated memory space, thus communication overhead is minimized. To equally distribute the work over the threads the subtrees need to contain a similar number of primitives, which is simply achieved by placing the first splits at the object median. The median can be approximated very fast by performing one streamed binning pass counting the primitives and then to place the split between bins where the primitive count is similar to the left and to the right. To improve scalability this first scanning can also be done in parallel by dividing the set of primitives among the threads, and merging their bins afterwards.

With their approach Shevtsov et al. [SSK07] demonstrated linear scalability with up to four threads, including kd-tree construction and rendering (see Figure 1).

5.2. Fast rebuild of adaptive SAH acceleration structure from scene graph

Through the combined use of several ideas that we summarize here, it is possible to build a high-quality acceleration structure (e.g. an approximate SAH kd-tree) at interactive frame rates without any substantive restriction on the kind

of motion. These ideas have been demonstrated in the Razor system to build a kd-tree acceleration structure [DHW*07] and could be adapted with minimal changes to other acceleration structures such as a BVH.

The key ideas are:

1. **Application stores scene geometry in a scene graph**. This scene graph serves as a "poor quality" bounding volume hierarchy that is updated every frame. (Razor does a complete update, but a lazy update is also possible).
2. **Build the acceleration structure lazily**. Only those parts of the kd-tree acceleration structure that are needed in the current frame are built. The lazy build relies on the scene graph to provide initial information about the organization of geometry in the scene.
3. **Use the hierarchy information from the scene graph to speed up construction**. When building the higher levels of the kd-tree, the builder uses bounding volumes from the scene graph hierarchy as proxies for all of the geometry they contain, making it very quick to build upper levels of the kd-tree. The Catmull-Clark patches at the leaf nodes of the scene graph provide an additional implicit hierarchy for their tessellated quads that is also used in the same way.
4. **Use a scan-based approximation to the SAH**. Hunt et al.'s scan-based approximation to the SAH is used to build the kd-tree [HSM06]. This fast builder is particularly important when the scene graph hierarchy is relatively flat and the sort required by a traditional kd-tree builder would be expensive.
5. **Use a specialized acceleration structure for mesh geometry**. Tessellated triangles from each surface patch are stored in their own simple bounding volume hierarchy. (If the number of triangles gets large, several such bounding volume hierarchies may be created for each patch). Each of these small bounding volume hierarchies is a leaf node in the kd-tree. Because the system has an implicit hierarchy for the triangles within a patch, the explicit bounding volume hierarchy hierarchy can be built very quickly yet serve as a very high quality acceleration structure.

It can be shown that several of these techniques – individually and in combination – *asymptotically* improve the performance of the build algorithm compared to standard techniques [HMFS07]. Furthermore, they are all complementary to each other; if any of them are disabled, overall build time increases [DHW*07].

Let n be the total number of "atomic" objects/polygons, v be the number of visible or nearly-visible objects/polygons, and assume that $v \ll n$. Traditional algorithms build the entire acceleration structure and require $O(n \log n)$ operations [WH06]. By using a "good" hierarchy from the scene graph or elsewhere (as defined in [HMFS07]), this cost can be reduced to $O(n)$. Intuitively, this improvement is due to the fact that the hierarchy is essentially a presort of the scene. If a hierarchy is used in conjunction with lazy evaluation, the operation count for the build is reduced to $O(v + \log n)$,

Multires / Lazy kd	Total patches	Touched patches	Touched triangles	Build time
No / No	543868	543868	21404	3.86 s
No / Yes	543868	27617	21404	0.131 s
Yes / No	543868	543868	1240978	4.50 s
Yes / Yes	543868	26655	1240978	0.705 s

Table 1: *Summary of kd-tree build performance with and without lazy build and multiresolution. All results use the scene graph hierarchy and a scan-based SAH approximation. The multiresolution setting tessellates patches to a specified maximum size (in this case, 64 pixels/triangle-vertex for eye rays) and uses a specialized acceleration structure for tessellated surface patches that is always instantiated lazily even when the kd-tree is not built lazily. These results are taken from Razor on a single core of a 2.6 GHz Intel Core 2 Duo, for frame 230 of the "Courtyard64" animation.*

which simplifies to $O(v)$ with the reasonable assumption that $\log(n) \ll v$. The scan-based approximation to the SAH does not change any of these asymptotic results as compared to a sort-based SAH builder when using hierarchy, but does provide better constant factors. For the case of lazy build *without* an initial hierarchy, the scan-based SAH does reduce the asymptotic operation count from $O(n \log n + v \log v \log v)$ to $O(n + v \log v)$, given certain other reasonable assumptions.

These operation counts do not include the cost of updating the scene graph hierarchy each frame. For coherent motion in a system with a "good" hierarchy and lazy updates, that cost is $O(\log n)$, so the total cost of scene graph update plus build remains $O(v + \log n)$, which is a major improvement over the $O(n \log n)$ cost without these techniques.

Table 1 shows that lazy build is very effective at reducing build time in a scene with high depth complexity. It also shows that when multiresolution patch tessellation is turned on (producing a large number of triangles), the use of a specialized acceleration structure for mesh geometry permits the build to remain relatively fast.

5.3. Handling semi-hierarchical motion using motion decomposition and fuzzy kd-trees

Although the presented methods to construct kd-trees are much faster than previous work they still need considerable time when the number of visible triangles is large.

Another approach that exploits coherence of motion in animations and dynamic scenes was introduced 2006 by Günther et al. [GFW*06]. Their *motion decomposition* technique builds on the idea of transforming rays instead of moving geometry [LAM01, WBS03].

Coherently moving parts of the scene are found automatically in a preprocessing step and affine transformations are computed to express the common motion. The residual motion is handled by a so called *fuzzy kd-trees*. Because the fuzzy kd-trees allow for small movement of its primitives they can be built in a preprocessing step and stay valid during

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

Figure 2: *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 submeshes 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.*

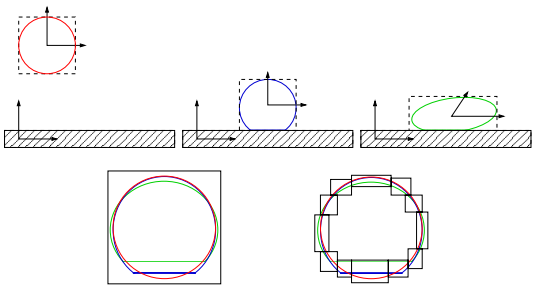

Figure 3: *Example of a motion decomposition. Top row: Three frames of an animation where a ball is thrown onto a floor, together with the bounding boxes and local coordinate systems of the two objects. The motion of these objects is encoded by affine transformations. Bottom row: Visualization of the bounded residual motion in the local coordinate system of the ball – coherent dynamic geometry is now "almost static". Note that the affine transformations can also compensate the shearing of the ball in the third frame yielding smaller fuzzy boxes.*

scene animation, almost completely avoiding the expensive reconstruction of kd-trees. At least for for skeleton-driven skinned meshes–by exploiting the present bone and skinning information, Günther et al. have shown that this limitation to predefined animations can also be removed [GFSS06].

5.3.1. Method overview

The concept and motivation of the motion decomposition framework is sketched in Figure 2. In the following we describe in more detail the individual parts.

Motion decomposition. The motion decomposition approach requires that the connectivity of the deformable mesh is constant and that the motion is semi-hierarchical. In particular Günther et al. assume that the motion is locally coherent. If this is the case they can decompose the motion into two parts: *affine transformation* and *residual motion*. Subtracting the affine transformation from the deformations yields a local coordinate system in which the (residual) motion of the vertices is typically much smaller (see Figure 3).

To find the affine transformations that approximately transform the coherent parts of a predefined animation to the next timestep, a linear least square problem on the vertex positions is solved [GFW*06]. For skinned animations this task is simpler: The affine transformations are directly provided from the application in form of bone transformations [GFSS06].

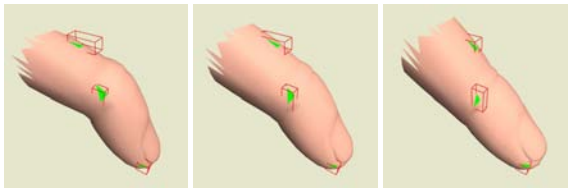

Figure 4: *The residual motion of each triangle (green) is bounded by a fuzzy box (red). Although the triangles move a little bit in the local coordinate system their fuzzy boxes do not change. As the fuzzy kd-tree is built over these fuzzy boxes instead of the triangles it is valid for* all *time steps.*

Fuzzy kd-tree. The residual motion is only in seldom cases zero (e.g. when there are only rigid-body transformations). To handle non-zero residual motion Günther et al. introduced fuzzy kd-trees. The residual motion of each triangle is bounded by a *fuzzy box*, a box bounding the motion of each vertex in the local coordinate system. A kd-tree is then built over the fuzzy boxes of the triangles instead of the triangles themselves, resulting in a fuzzy kd-tree. As long as the fuzzy boxes are not violated by too strong residual motion the fuzzy kd-trees stay valid even during mesh animations (see Figure 4). Thus fuzzy kd-trees can be built in a preprocessing phase and do not require rebuilding.

For predefined animations, conservative fuzzy boxes can be computed in a preprocessing step [GFW*06]. For skinned animations, the fuzzy boxes are found by sampling the pose space of the skinned mesh [GFSS06].

Two-level kd-trees. Because the relationship between the coherently moving parts of an animation and their motion is determined by affine-only transformations Günther et al. use a two-level acceleration structure in the spirit of [LAM01] and similar to [WBS03]. For each frame to be rendered they update the transformations and current bounding boxes of objects having an own fuzzy kd-tree. Then a small top-level

kd-tree is built over these bounding boxes. Only this top-level kd-tree needs to be rebuilt every frame, which can be done very quickly because the number of moving objects is usually small (less than 100), allowing interactive frame rates.

Ray traversal works as in [WBS03] by transforming the rays into the local coordinate system of objects encountered in top-level leaves and continuing traversal of the corresponding fuzzy kd-tree.

5.3.2. Clustering of coherently moving primitives

Motion decomposition is only efficient when it operates on coherently moving primitives. However, there are often numerous parts of a dynamic scene that move independently. Thus it is necessary to identify and to separate these parts, which is performed automatically in a preprocessing step by specialized clustering algorithms. The measure to guide these clustering algorithms is the residual motion, which should be minimized to get efficient fuzzy kd-trees.

Predefined animations. To partition the primitives of a predefined animation into clusters Günther et al. apply a generalized Lloyd relaxation [Llo82, DFG99, GG91] algorithm. In each iteration step they firstly find affine transformations that minimize the residual motion of each cluster and subsequently reassign each primitive to the cluster where its residual motion is smallest. The iteration process stops when the clustering converged, i.e. when no primitive change its cluster anymore.

As Lloyd relaxation is prone to find local minima and the optimal number of clusters is not known in advance, they start with one cluster and iteratively insert a new cluster until the cost function converges. When inserting a new cluster it is initialized with few seed primitives – the primitives with the largest residual motion. The already existing clusters are also newly initialized with seed primitives having the *smallest* residual motion in each cluster. These seed primitives act as prototypes of the common motion of the (currently) clustered primitives and ensure a stable clustering procedure.

Skinned meshes. For skinned meshes the clustering is simpler because the potential clusters are already given in form of the bones. By sampling the pose space Günther et al. assigned each primitive to the bone/cluster where its residual motion is smallest. 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.

5.3.3. Results and discussion

Using the motion decomposition approach, interactive frame rates have been reported for both animations [GFW*06] and on-the-fly skinned meshes [GFSS06]. (see Figure 5. Ray tracing with a top-level kd-tree and fuzzy kd-trees is about

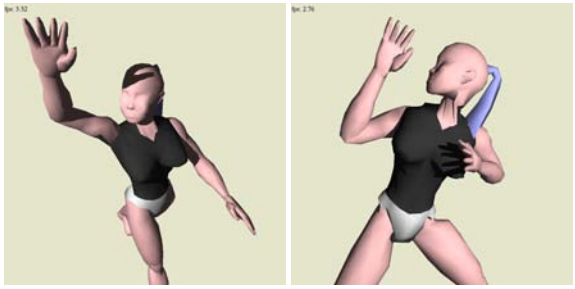

Figure 5: *"Cally" featuring self shadowing in interactively changeable poses, ray traced with 4 frames per second at 1024^2 pixels on a single 2.4 GHz AMD Opteron [GFSS06].*

2× slower compared to using a kd-tree optimized for one single time step (that does not support dynamic changes). This overhead is caused by (1) the reduced culling efficiency of the kd-trees due to overlapping object bounding boxes and/or overlapping fuzzy boxes, (2) the time needed to rebuild the top-level tree, and (3) the more complex ray traversal that includes the transformation of rays between coordinate systems.

However, using a top-level acceleration structure also provides additional benefits. It is easy to instantiate one object several times. Furthermore, the local coordinate system of the axis-aligned fuzzy kd-trees can be rotated to achieve better bounds and thus performance.

The motion decomposition approach is restricted to semi-hierarchical, locally coherent motion – handling random motion is not supported. Furthermore, the help of the application is needed to provide additional information such as bone and skinning information.

These motion clustering methods can also provide topological information to guide the construction of other acceleration structures, e.g. for a BVH to minimize degeneration after bounding box updates due to moving primitives.

6. BVH-based approaches

As mentioned before, kd-trees have enjoyed great popularity in real-time ray tracing [WSBW01, RSH05, SWS02], and have even once been declared the "best known method" for fast ray tracing [Sto05]. While a strong contender for the most efficient data structure with respect to rendering time, they also have a number of drawbacks, partially addressed above. First among these is that kd-trees are not particularly well suited to dynamic updates, because even small changes to the scene geometry typically invalidate the tree. Consequently, kd-tree-based ray tracers must typically build a new kd-tree from scratch for every frame, either using the fast build techniques described in Section 5.1, or a lazy/on-demand build scheme as outlined in Section 5.2.

As also argued above, bounding volume hierarchies differ from kd-trees in that a BVH is a hierarchy built over the *primitives*, with bounding information stored at each node

in the hierarchy. The beauty of BVHs with respect to dynamic scenes is that they are much more flexible in terms of incremental updates as has been heavily exploited in the collision detection community [JP04, LAM05, vdB97, TKH*05, LM04]. For deformable scenes, just re-fitting a BVH – i.e., recomputing the hierarchy nodes' bounding volumes, but not changing the hierarchy itself – is sufficient to produce a valid BVH for the new frame. BVHs also allow for incremental changes to the hierarchy [YCM07, EGMM06] (see below), and referencing each primitive exactly once greatly simplifies the build [WBS07, WK06, EGMM06].

Though a BVH's advantages with respect to dynamic scenes are well known, many researchers previously assumed that a bounding volume hierarchy could not be competitive with either kd-trees or grids with respect to render time [HPP00, ML03][§]. Consequently, BVHs have long been ignored in real-time ray tracing, until their advantages with respect to dynamic updates led to renewed interest. However, in practice properly constructed BVHs achieve quite competitive performance even for single ray code. Beyond just building strategy, many of the techniques originally developed for tracing packets or frusta with kd-trees are also applicable to BVHs [LYTM06, WBS07]. In the remainder of this section, we will first discuss how to quickly traverse BVHs, and then discuss the various approaches to using them for animated scenes.

As for kd-trees, traversal performance for a BVH depends on two parts: the effectiveness of the actual hierarchy, and the efficiency of the traversal methods. In principle, BVHs can use arbitrary branching factors, and arbitrarily shaped bounding volumes. In practice, however, BVHs are usually binary trees of axis-aligned bounding boxes (AABBs) and we will focus on these.

6.1. Building effective BVHs

The effectiveness of a BVH depends on what actual hierarchy the build algorithm produces. Like for kd-trees, best results seem to be achieved using top-down, surface area heuristics-based builds. The earliest method specified the hierarchy by hand [RW80], but this was soon replaced by split in the middle with cycling axes [KK86]. Both techniques can result in rather inefficient BVHs, and realistically only the automatic generation of a BVH is feasible.

Goldsmith and Salmon proposed the use of a cost estimate to minimize a BVHs expected traversal cost [GS87]. This was later refined by MacDonald and Booth [MB89] into a greedy build for kd-trees. The original Goldsmith and Salmon algorithm built a tree incrementally by successively inserting primitives into a tree, and letting those "trickle down" into the subtrees with minimum expected cost.

[§] Arguably, the reason these studies indicated inferior performance is that these studies only considered BVHs built with Goldsmith Salmon-like, bottom up build strategies, which tend to perform worse than kd-tree like top-down builds (also see [Hav07]).

Today, the state of the art in building effective BVHs is to use SAH top-down builds similar to the way SAH kd-trees are built. Applying an SAH build to BVHs was first proposed by Müller and Fellner [MF99], but unfortunately that result was not widely appreciated until recently. The same is true for related work by Masso et al. [ML03].

Following [WBS07], a pseudo-code implementation for a top-down SAH build (with $O(N \log^2 N)$ runtime complexity) is detailed in Algorithm 1. The BVHs built with this algorithm are essentially the same as the ones built by Müller's method except that Algorithm 1 uses the centroid of the primitive's bounding box to decide on which side of a "split plane" to place a primitive instead of using the axis aligned "edges" of the primitive. Note, however, that even though the decision on which side to place a polygon is based on its centroid, the cost evaluation uses the correct bounding volumes over the primitives, not over the centroids.

Algorithm 1 Centroid-based SAH partitioning

```
function partitionSweep(Set S)
  bestCost = T_tri*|S| {cost of making a
  leaf}
  bestAxis = -1, bestEvent = -1
  for axis = 1 to 3 do
    sort S using centroid of boxes in cur-
    rent axis
    {sweep from left}
    set S1 = Empty, S2 = S
    for i = 1 to |S| do
      S[i].leftArea = Area(S1) {with
      Area(Empty) = ∞}
      move triangle i from S2 to S1
    end for
    {sweep from right}
    S1 = S, S2 = Empty
    for i = |S| to 1  do
      S[i].rightArea = Area(S2)
      {evaluate SAH cost}
      thisCost = SAH(|S1|, S[i].leftArea,
      |S2|, S[i].rightArea)
      move Triangle i from S1 to S2
      if thisCost < bestCost  then
        bestCost = thisCost
        bestEvent = i
        bestAxis = axis
      end if
    end for
  end for
  if bestAxis = -1 then {found no partition
  better than leaf}
    return make leaf
  else
    sort S in axis 'bestAxis'
    S1 = S[0..bestEvent); S2 =
    S[bestEvent..|S|)
    return make inner node with axis
    'bestAxis';
  end if
end
```

Figure 6: *Two screenshots from Lauterbach et al.'s BVH based ray tracing system. Left: One frame from a 40K triangle dress simulation animation, running at 13 frames per second (512 × 512 pixels, one point light) on a dual 2.8 GHz Pentium IV PC. Right: a fracturing bunny including shadows and reflections, running at an average of 6 frames per second on same hardware.*

6.2. Fast BVH traversal

After the BVH is built, there are many factors that determine the overall traversal speed. Given AABBs as bounding primitives, one of the first choices is which ray-box test to use. A variety of tests exist [KK86, MW04, WBMS05] but most systems today seem to use the SLABS algorithm [KK86] (also see [BWS06]). In addition, a number of useful optimizations for single ray code related to tree layout or traversal order have been proposed [Hai91, Smi98, Mah05]. Most of these optimizations are still useful to packet traversal.

6.2.1. Packet tracing

The use of packets for BVH traversal was proposed independently by Wald et al. [WBS07] and Lauterbach et al. [LYTM06]. Both demonstrated that by using packets, BVHs can be competitive with kd-trees for raw rendering performance. Lauterbach's system originally used SIMD packet traversal with 2×2 rays per packet, but later implemented a frustum method similar to Reshetov et al. [RSH05] for higher performance – depending on scene complexity – for packets of up to 16×16 rays (also see Figure 6).

6.2.2. DynBVH packet-frustum traversal

In addition to SIMD packet tracing proposed by Lauterbach et al., Wald et al. [WBS07] proposed a modified traversal algorithm that can lead to higher amortization – and ultimately, higher performance – than that afforded by SIMD alone. The algorithm proposed in [WBS07] essentially combines four independent algorithmic optimization into one new traversal algorithm: a speculative "first hit" descend, first active ray tracking, interval arithmetic-based culling, and SIMD processing for all operations.

Early hit test with speculative descent. In a standard packet tracer all rays are tested at each tree node (albeit possibly 4 at a time if SIMD instructions are used). In principle, there is no need to test all rays, as even a single hit requires recursion into that respective subtree. For kd-trees or bounding plane hierarchies, the fastest known traversal algorithms track the active ray intervals [Hav01, Wal04], and thus have to process all rays even if recursion is already required. For a BVH, however, this is not the case, and upon any successful intersection, the packet can immediately enter this subtree without considering any of the remaining rays.

By having all rays in a packet descend to the two children when the first ray hits the parent, the $N \times N$ (packet size) ray-box tests are often replaced with only a single test. This comes at the cost of some rays (speculatively) descending that miss the parent, but this is no different from a pure SIMD packet traversal.

Tracking the first active ray. As just described, as soon as any ray in a packet hits the box they all descend. However, the ray that hits the box may not be the first ray in the packet. Moreover, a ray that has missed a node will also miss all child nodes. Packets can take advantage of this by not testing rays that have already missed an ancestor of the current node. This is easily accomplished by storing the index of the first ray that has not yet missed an ancestor and starting the loop over rays at that index. Packets still immediately descend as soon as a hit is detected.

Early miss exit. The combination of early hit tests and first active ray tracking essentially makes those cases in which

Algorithm 2 Pseudo-code for the fast packet/box intersection. Both "full hits" (i.e., first ray that hits parent also hits box) and "full misses" (i.e., a covering frustum misses the box) are very cheap, and have a constant cost independent of packet size. Only for rays partially hitting the box does the method perform more than the first two cheap tests.

```
{Compute ID of first ray hitting AABB box}
{'first' is the ID of the first ray hitting
box' parent}
function findFirst(ray[maxRays], int first,
AABB box)
 {First: Quick "hit" test using 'first'
 ray}
 if ray[parentsFirstActive] intersects box
 then
  {first one hits → packet hits...}
  return parentsFirstActive
 end if

 {Second: Quick "all miss" test using ei-
 ther frustum or interval arithmetic}
 if frustum(ray[0..N]) misses box then
  return maxRays {all rays miss}
 end if

 {Neither quick test helped, test all rays}
 for i = parentsFirstActive .. do
  if ray[i] intersects box then
   return i {all earlier ones missed}
  end if
 end for
 return maxRays {all rays have missed}
end
```

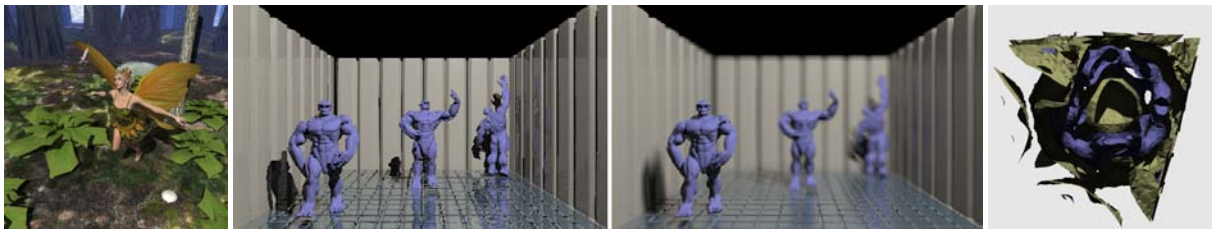

Figure 7: *Some examples from systems using the DynBVH traversal algorithm. a) The Utah Fairy model (180k triangles), with textures and shadows from one point light, running at 3.7 frames per second (including BVH updates) on a 2.6 GHz dual-core Opteron PC [WBS07]. b+c) The same system, extended to handle Whitted-style ray tracing and distribution ray tracing [BEL*07], though running at much lower frame rates (largely due to the higher number of rays to be traced). d) An extension of that system to ray tracing iso-surfaces of tetrahedral meshes, showing a 225k tet "buckyball" running at 42 frames per second on a 2.4 GHz 16-core Opteron workstation.*

the packet actually overlaps the box very cheap. However, if the packet misses the box, all the rays in the packet would still be tested to find that none of them hits the box.

For this case, BVHs can employ a similar idea as proposed for kd-trees by Reshetov et al. [RSH05]. Using interval arithmetic, an approximate (but conservative) packet-box overlap test can be performed. This conservative tests can immediately signal that the traversal should skip further tests as no rays in the packet can possibly intersect the box. Instead of interval arithmetic, one can also use actual geometrical frusta [RSH05, BWS06], the efficiency is similar.

The first hit test and early miss test can also be combined. Combining the two inexpensive tests allows the traversal to usually determine whether or not the packet should descend with at most two tests for a full $N \times N$ packet. It is important to note that the two tests cover orthogonal cases and so the order in which they are applied is not critical.

Testing the remaining rays. If both the first hit test and the conservative miss test failed, the remaining rays in the packet are intersected until one is found to hit. The pseudo-code for the resulting packet-box intersection test is given in Algorithm 2. Compared to the other two cases that have constant cost, testing the remaining rays is linear in the number of rays in the packet. Though implemented in SIMD the test can be quite costly. Fortunately, this case happens rarely as shown empirically in Section 6.2.3.

Leaf nodes. When reaching a leaf node, the triangles in that leaf get intersected. In addition to the first active ray, the original DynBVH system also computed the last active ray every time a leaf was reached, and intersected only all rays inbetween. In some cases, higher performance can be reached by testing all rays against the box, and skipping triangle intersections for inactive rays [WFKH07]. Obviously, the triangle intersections are performed in SIMD batches of four, and with SIMD frustum culling [DHS04, BWS06].

6.2.3. DynBVH performance

Using this traversal algorithm, the fast early hit and frustum exit tests can reduce the number of ray-box tests by roughly an order of magnitude compared to a pure packet traverser [WBS07] (Table 2).

scene	(A) early hit exits	(B) frustum exits	(C) last resort packet test
erw6	52.3%	42.9%	4.8%
conference	51.9%	35.3%	12.8%
soda hall	49.5%	27.5%	23.0%
toys	49.7%	32.2%	18.1%
runner	44.1%	25.3%	30.6%
fairy	49.1%	30.2%	20.7%

Table 2: *Relative number of cases where Algorithm 2 can immediately exit after the first test, after the second test, and during the loop over all rays respectively for ray casting with 16×16 rays per packet. Data from [WBS07].*

In combination, algorithmic optimizations and SSE processing allow the DynBVH system to achieve performance that is quite interactive, and roughly on par with the best known performance for kd-trees (see Table 3). The actual impact of the algorithm depends heavily on scene and ray distribution: As the traversal algorithm exploits coherence, its performance somewhat suffers for packets with lower coherence and for scenes with subpixel-sized geometry, and performance in those cases can deteriorate to that of a pure packet traverser.

Extensions. In its original publication, the DynBVH algorithm was presented only for triangle meshes, for relatively simple shading and ray distributions, and for a refitting-only approach to handling changing geometry. Since then, Bou-

Scene	anim.	#tris	OpenRT Pentium4 2.4 GHz	MLRTA Xeon 3.2 GHz w/ HyperThr.	BVH Opteron 2.6 GHz
erw6	no	800	2.3	50.7	31.3
conf	no	280k	1.9	15.6	9.3
soda	no	2.5M	1.8	24	10.9
toys	yes	11k	–	–	21.9
runner	yes	78k	–	–	14.2
fairy	yes	180k	–	–	5.6

Table 3: *Performance in frames/sec. (at 1024^2 pixels, single 2.6 GHz Opteron CPU, including simple shading) compared to the OpenRT and MLRTA systems. Performance data is taken from [WBS07], and is for a variety of both static and animated models detailed in that publication.*

los et al. [BEL*07] have investigated how the algorithm would behave for more realistic ray distributions like those produced by either Whitted-style recursive ray tracing and Cook-style distribution ray tracing (see Figure 7b+c). They found that the algorithm worked quite well even for these non-primary ray distributions, and that both Whitted- and Cook-style ray tracing achieved similar number of rays per second as ray casting with shadows (i.e. within a factor of 4). Although the increase in the total number of rays in a frame resulted in lower absolute frame times. It should be noted that Boulos et al. disabled several optimizations in their system that only apply to ray casting or ray casting with shadows (RCS) that were used in the original paper [WBS07]. This differences, including differences in the shading model and normalized directions for camera rays, account for their lower performance for RCS.

As compared to secondary ray tests by Reshetov [Res06], the Boulos et al. [BEL*07] system retained SIMD benefits for several bounces of reflection. In addition to a SIMD benefit, the traversal algorithm still provided further speedup beyond a simple SIMD benefit alone.

While the original DynBVH system relied exclusively on refitting, the current code base also includes fast from-scratch rebuilds in the spirit of Wächter et al. [WK06] and Shevtsov et al. [SSK07] (see below), as well as a and a hybrid approach that mixes refitting and fast, asynchronous rebuilding [IWP07]. A CELL variant of the DynBVH algorithm was presented by Benthin et al. [BWSF06], which through low-level, architecture-specific optimizations achieved up to 8× the performance of the CPU-based variant, up to a total of 231.4 frames per second (for ray casting the erw6 scene) on a single 2.4 GHz CELL processor.

Apart from its application to triangle meshes, the DynBVH algorithm was also applied to interactively visualizing iso-surfaces in tetrahedral meshes [WFKH07] (Figure 7d). It allowed for handling time-varying data, and achieved interactive frame rates even for highly complex models.

6.3. Variations and hybrid techniques

In the previous section, we have focused on the DynBVH system, as its traversal algorithm gives it a speed advantage over other, single ray or purely packet-based systems. However, a large number of possible variations exist, most of which are orthogonal to the techniques used above. For example, Yoon et al. [YM06] have proposed cache-oblivious data layouts that improve memory efficiency (similar techniques exist for kd-trees [Hav97]); Mahovsky et al [MW06] proposed a Q-Splat like compression approach to reduce the BVH's memory footprint. Other variations that have not been properly investigated yet include the use of BVHs with higher branching factors than two, or BVHs with non-AABB bounding volumes, both of which may be beneficial.

SKD-Trees. One particular popular variation is to encode the BVH using a hierarchy of bounding planes: instead of storing a full 3D bounding box in each node, each node stores a set of two parallel planes that partition the current node's bounding volumes into two (potentially overlapping) halves. The resulting data structure looks similar to kd-trees but behaves like a bounding volume hierarchy. In particular, the data structure allows for refitting like a BVH does, and, while known outside graphics since Ooi's paper from 1987 [OSDM87], for ray tracing dynamic scenes were proposed independently by Havran et al. [HHS06], Woop et al. [WMS06], and Wächter et al. [WK06] (albeit under three different names, calling them skd trees, b-kd trees, and bounding interval hierarchies, respectively). Though their data structure is similar in nature, there are several important differences in the focus of each of those papers.

Wächter et al. focused on fast building and presented an $O(N)$ algorithm for building the tree. Their build strategy only uses spatial median builds that may prove inefficient. Woop et al. [WMS06] use a four-plane hierarchy in each node (two planes per child) that leads to a slightly different data structure; apart from that, they focus on using this technique to support dynamic scenes on a ray tracing hardware architecture that was originally built for kd-trees [SWS02,WSS05]. The paper by Havran et al. [HHS06] follows Ooi's originally proposed skd-trees, but goes a step further, and proposes H-trees that consist of two splitting planes: bounding nodes and splitting nodes (skd-tree nodes). Bounding nodes are put only optionally based on an approximative SAH cost model. The paper also contains the description of AH-trees that resembles the use of radix sort for spatial data structures with construction time $O(N \log\log N)$.

How bounding plane hierarchies compare to traditional BVHs has not been fully investigated, yet. On the positive side, bounding plane hierarchies are similar to kd-trees, and allow for using traditional kd-tree traversal algorithms with a minimum of modifications. They also have a smaller node layout than BVHs, but are as fast to build and as easy to update. On the downside, traditional BVHs will yield somewhat tighter bounding volumes (since they bound each subtree in 3 dimensions, not only one), do not have to deal with splits producing "empty" subtrees, and feature somewhat simpler traversal codes (because the traversal does not have to track per-ray overlap intervals). Lauterbach et al. [LYTM06] report roughly comparable performance for his BVH implementation than for a reference skd-tree implementation; Wald et al.'s BVH-based system [WBS07] is much faster than either, but uses a different traversal algorithm that could also be applied to skd-trees.

6.4. Fast BVH building and updating

Like any data structure, BVHs become invalid if the underlying geometry undergoes motion. Though their suitability to refitting was one of the main reasons for using BVHs, refitting only works for deformable meshes. While refitting may cover an important class of applications, no practically relevant ray tracer will be able to rely on refitting alone.

In practice, both BVHs and kd-trees are binary, axis-aligned, and hierarchical data structures that are ideally built

using top-down SAH strategies. As such, most of the trade-offs and techniques for handling rebuilds and updates for kd-trees (see previous Section) apply similarly to BVHs. In particular, any or all of the techniques used in the Razor framework (scene graph-guided rebuild, lazy construction, and multiresolution geometry) would be applicable without major changes; the same is true for approaches like motion decomposition (Section 5.3), rigid-body animation through local-coordinate transformations (Section 4.3), and fast, approximate SAH builds (Section 5.1).

6.4.1. Incremental updating

As already mentioned before, if the scene only contains *deformable* geometry, a BVH can be updated to reflect the new positions of the geometry. While this can lead to deterioration of the BVH, a BVH chosen over a range of animation can help to reduce this deterioration [WBS07]. While this deterioration can be reduced for either known animations or for ranges of motion similar to known poses, other kinds of motion require more general techniques.

Periodic rebuilding. Rebuilding per frame is the most general, but also most expensive; refitting is extremely fast, but can lead to excessive BVH deterioration. Instead of relying on only one of these techniques, they can be combined. This was first proposed by Lauterbach et al. [LYTM06], who used fast refitting most of the time, and only rebuilt the BVH from scratch whenever a given cost heuristic indicated excessive quality deterioration of the BVH. This lowered the average frame time, sometimes significantly; however the frame waiting for the rebuild to finish would take much longer to complete, leading to a disruptive pause. Ize et al. [IWP07], handle the refitted BVH deterioration without introducing disruptions by using a fraction of the computational resources for rebuilding a BVH asynchronously while the rest of the computer renders and refits the current BVH. Then when the asynchronously built BVH is ready, it can be used for rendering and refitting while a new BVH is rebuilt. This takes advantage of the growing parallelism in multi-core architectures.

Incremental hierarchy updates. Instead of always rebuilding BVHs from scratch, incremental updates should suffice for coherent motion. Since each node in a BVH is referenced exactly once, incremental updates are rather simple: an entire subtree can be moved by updating a few pointers and refitting the affected nodes and some of their ancestors. The tricky part lies in determining which updates to perform.

The basic idea for this approach was also proposed by Lauterbach et al. [LYTM06], who proposed to selectively re-build those parts of the hierarchy that his rebuild heuristic flagged as deformed. However, this did not affect nodes higher up in the tree, so unsatisfactory results were reported. More recently, Yoon et al. [YCM07] presented an approach to selectively update BVHs that could also handle highly complex scenes with strong deformations: based on a heuristics that essentially measures the overlap of subtrees, the algorithm recursively finds – and fixes – pairs of BVH nodes whose overlap is a) high, and b) can be reduced through swapping their subtrees. Keeping these pairs in a priority queue, the algorithm always fixes the worst deformations first, and can, in particular, be given a fixed time budget. The generated trees will likely not be as efficient as those built from scratch (in particular, the generated trees are not the same as those built by a surface area heuristic), but the overall performance for complex dynamic scenes has been shown to be faster than using a rebuild heuristic [YCM07].

6.4.2. Fast BVH building

Refitting and incremental updates can handle a wide range of different models, but are inherently limited to deformable models. For other cases, full rebuilds are required. The BVH build algorithm mentioned in Section 6.1 (Algorithm 1) serves as a baseline for the state of the art in building *good* BVHs. Though much faster than corresponding SAH-builds for kd-trees (compare, e.g., [WBS07] and [WH06]), it is still non-interactive except for trivial models. As for kd-trees, there is a trade-off between build quality and build time, and significantly faster build times are possible if a certain redic-tion in BVH quality is allowed.

BIH-build: Fast spatial median build. Probably the fastest way of building BVHs known today is an adaption of the fast spatial median split introduced by Wächter et al. in their Bounding Interval Hierarchy paper [WK06]. While this algorithm was originally proposed for two-plane hierarchies, it can be applied directly to BVHs. Just like Algorithm 1, the BIH-style build works only on the centroids, not on actual primitives, but instead of building with an expensive cost function, it successively splits along the spatial median until less than a threshold number of primitives per leaf is reached. A key point of the algorithm is to compute the centroids' bounding box only once at the beginning, and then to successively subdivide this box just like a spatial subdivision would do (i.e., without ever re-computing a sub-tree's bounding box until the BVH is fully built); this not only results in a very regular subdivision that could be built in $O(N)$ complexity, it also seems – at least in our experience – to result in better trees than those produced by successively splitting the bounding box of each subtree, though the exact reasons for this are still somewhat unclear.

As no cost function is used, the resulting BVHs are less efficient than BVHs built with Algorithm 1 (see Table 4), but the exact performance impact depends on the scene. Note that another $O(N)$ algorithm for building BVHs has been proposed independently by Eisemann et al. [EGMM06], but real-time data is not available for this algorithm.

Binned SAH builds. In addition to the BIH-build, it is also possible to implement an SAH-based binning strategy as discussed above for kd-trees (Section 5.1). Since no fast BVH build numbers have been published, yet, we have implemented a preliminary version of this algorithm, which works almost exactly as proposed by Shevtsov et al. [PGSS06],

		build time (abs)			render perf (rel)	
scene	#tris	sweep	BIH	binned	BIH	binned
fairy	174K	893 ms	33 ms	100 ms	80%	97%
conf	282K	1.4 s	60 ms	172 ms	66%	90%
buddha	1.08M	6.5 s	255 ms	469 ms	79%	88%
thai statue	10M	85 s	3.3 s	6 s	84%	102%

Table 4: *Build times for a baseline SAH sweep build, a BIH-style spatial median build, and a binned SAH build; as well as impact on render time relative to the SAH sweep build. Impact on render performance corresponds to the DynBVH system with packets of 8×8 rays, and may vary for other traversal algorithms; absolute build times are for a single thread running on a 2.6 GHz Xeon 5355 (Clovertown).*

except for two modifications: Instead of binning the actual primitive extends as required by a kd-tree, we can again use only the centroids for the build; we therefore start with a bounding box for the centroids, and subdivide that into N equally-sized bins B_i along the bounding box' axis of major extent. We then project each primitive's centroid into its respective bin, and compute, for each bin, the number of primitives projecting to it, as well as the actual geometry bounds of all primitives associated with that bin; and evaluate the SAH for the $N-1$ possible partitions into $B_0..B_{i-1}$ and $B_i..B_{N-1}$, and take the partition with lowest cost.

As can be seen from Table 4, for a reasonable number of bins ($\sim 8-16$) this build provides nearly the same build performance as the BIH-style build, while providing nearly the same quality as the full SAH build. In general, it seems that like for traversal, most of the techniques developed for kd-trees work just as well for BVHs; in this case, the build is even simpler for BVHs, as trees as generally shallower (i.e., less splits to be determined), multiple references are not allowed, and each partitioning can be done strictly "in place" in a quicksort-like algorithm (also see [Hav07]). Being only a preliminary experiment, a more highly optimized implementation might yield even better performance, and parallelization was not fully implemented yet at all.

6.5. BVH-based approaches – Summary

Though long neglected, BVHs have recently regained favor. While this was mostly due to their capabilities for refitting, it now appears that BVHs in general perform well once the same effort is made to optimize them as was invested in kd-tree traversal. On comparable hardware, BVH performance still lags somewhat behind the fastest published performance for kd-trees (Reshetov's MLRT system [RSH05]), but not conclusively so. With respect to dynamic scenes, kd-trees seem have received more attention than BVHs (Razor, approximate SAH-builds, ...), but most of these techniques generalize to BVHs, which are arguably somewhat more flexible with respect to updating and building. In most other respects (memory consumption, traversal algorithms, suitability for packets, frusta, complex scenes, ...), BVHs and kd-trees are very competitive.

7. Grid-based approaches

In this section we present techniques for traversing and building grid acceleration structures.

We will begin by contrasting the grid acceleration structures with kd-trees and BVHs. While both kd-trees and BVHs are hierarchical, adaptive data structures, grids fall into the category of uniform spatial subdivision. The trade-off between uniform and adaptive subdivision has been discussed in Section 3.1.3: adaptive subdivision often works better for complex scenes with uneven geometry distributions, but are generally harder to build. As evident from the previous two sections, several techniques for fast building of adaptive data structures have been developed, and at least for special cases like deformable motion, refitting and incremental updates can be really fast. Nevertheless, building hierarchical data structures is still significantly more costly than simply "rasterizing" the triangles into a regular grid, which is conceptually similar to a radix sort [Knu98]. Building a regular grid can be done in a single pass, in parallel [IWRP06], and is roughly as fast for complete rebuilds as simple refitting is for BVHs.

Being able to rebuild from scratch every frame, the grid does not need to make any assumptions on the kind of motion. Despite these advantages, grids until recently received little attention.

7.1. Parallel grid rebuilds

Rebuilding a grid acceleration structure consists of three main steps: clearing the previous grid cells and macro cells of previous triangle references; inserting the triangles into the grid cells that they intersect; and building the macro cells. We specifically handle the clearing of the grid ourselves, rather than throwing it away and creating a new data structure for two reasons: firstly, this lowers the number of expensive memory allocations/deallocations required; and secondly, by iterating through the previous macro cells, we can quickly find the grid cells that require clearing, rather than going through all grid cells. Ize et al. [IWRP06] contain more details on the cell clearing and macro cell building steps, and how they can be trivially parallelized.

Ize et al. [IWRP06], parallelized the triangle insertion by recognizing it is equivalent to triangle rasterization onto a regular structure of 3D cells, and then showing that a sort-middle approach works better than sort-first and sort-last approaches [MCEF94]. In their sort-middle approach, each thread performs a coarse parallel bucket sort of the triangles by their cell location. Then each thread takes a set of buckets and writes the triangles in those buckets to the grid cells. Since each thread handles writing into different parts of the grid, as specified by the buckets, there is no chance of multiple threads writing to the same grid cells; thus expensive mutexes are not required.

This sort-middle method is relatively straightforward to load balance, each triangle is read only once, and there are no write conflicts. There is no scatter read nor scatter write,

which is good for broadband/streaming architectures. The disadvantage is that it requires buffering of fragments between the two stages, however in practice the buffering can be implemented efficiently, and was shown to produce almost no overhead.

7.2. Coherent grid traversal

Implementing ray packets for a grid is not as straight forward as for the tree-based acceleration structures. The primary concern with packetizing a grid is that with a 3DDDA, different rays may demand different traversal orders. Wald et al. [WIK*06] solve this by abandoning 3DDDA altogether, and propose an algorithm that traverses the grid *slice by slice* rather than cell by cell. For example, the rays in Figure 9 can be traversed by traversing through vertical slices; from cell A in the first slice, the rays are traversed to cells B and D in the second slice, then to C and E in the third, and so on. In each slice, all rays would intersect all all of the slice's cells that are overlapped by any ray. This may traverse some rays through cells they would not have intersected themselves, but will keep the packet together at all times. In practice, ray coherence easily compensates for this overhead.

The rays are first transformed into the canonical grid coordinate system, in which a grid of $N_x \times N_y \times N_z$ cells maps to the 3D region of $[0..N_x) \times [0..N_y) \times [0..N_z)$. In that coordinate system, the cell coordinates of any 3D point p can be computed simply by truncating it. Then, the dominant component (the $\pm X$, $\pm Y$, or $\pm Z$ axis) of the direction of the first ray is picked. This will be the *major traversal axis* that we call $\vec{K}$; all rays are then traversed along this same axis; the remaining dimensions are denoted $\vec{U}$ and $\vec{V}$.

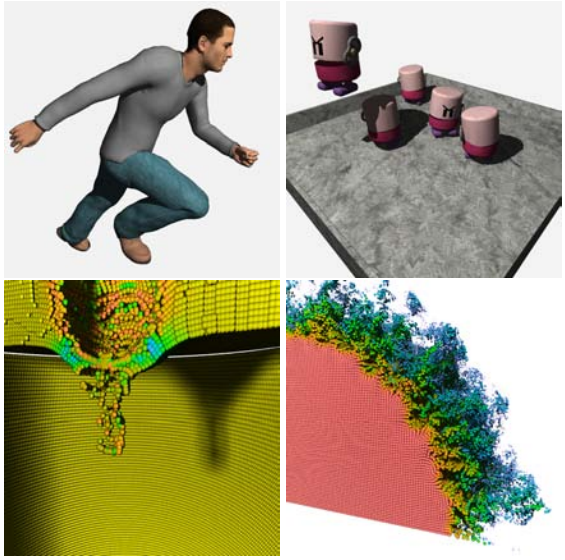

Figure 8: *Examples of dynamic scenes (full rebuilds occur every frame) that benefit from coherent grid traversal: The running character model (78k triangles) and the animated wind-up toys (11K triangles) that walk and jump incoherently around each other respectively achieve 7.8 and 10.2 frames per second on a dual 3.2 GHz Xeon with shading and hard shadows. The next two images show 213k sphere and 815k sphere scientific visualizations of particle data sets [GIK*07] and achieve 8.8 and 6.9 frames per second with area lights sampled 16 times per hitpoint when run on a 16 core shared memory 2.4 GHz Opteron system.*

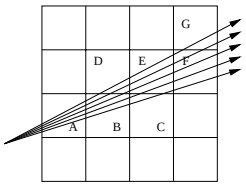

Figure 9: *Five coherent rays traversing a grid. The rays are initially together in cells A and B, but then diverge at B where they disagree on whether to first traverse C or D in the next step. Even though they have diverged, they still visit common cells (E and F) afterwards.*

Now, consider a slice k along the major traversal axis, $\vec{K}$. For each ray r_i in the packet, there is a point p_i^{in} where it enters this slice, and a point p_i^{out} where it exits. The axis aligned box $\mathcal{B}$ that encloses these points will also enclose all the 3D points – and thus, the cells – visited by at least one of of the rays. Once $\mathcal{B}$ is known, truncating its min/max coordinates yields the u,v extents of all the cells on slice k that are overlapped by any of the rays (Figure 10d).

Extension to frustum traversal. Instead of determining the overlap $\mathcal{B}$ based on the entry and exit points of *all* rays, one can compute the four planes bounding the packet on the top, bottom, and sides. This forms a bounding frustum that has the same overlap box $\mathcal{B}$ as that computed from the individual rays. Since the rays are already transformed to grid-space, the bounding planes are based on the minima and maxima of all the rays' u and v slopes along $\vec{K}$. For a packet of $N \times N$ primary rays sharing a common origin, these extremal planes are computed using the four corner rays; however for more general (secondary) packets all rays must be considered.

Traversal Setup. Once the plane equations are known, the frustum is intersected with the bounding box of the grid; the minimum and maximum coordinates of the overlap determine the first and last slice that should be traversed. If this interval is empty, the frustum misses the grid, and one can terminate without traversing.

Otherwise, one computes the minimum and maximum u and v coordinates of the entry and exit points with the first slice to be computed. Essentially, these describe the lower left and upper right corner of an axis-aligned box bounding the frustum's overlap with the initial slice, $\mathcal{B}^{(0)}$. Note that one only needs the u and v coordinates of each $\mathcal{B}^{(i)}$, as the k coordinates are equal to the slice number.

Incremental traversal. Since the overlap box $\mathcal{B}^{(i)}$ for each slice is determined by the planes of the frustum,

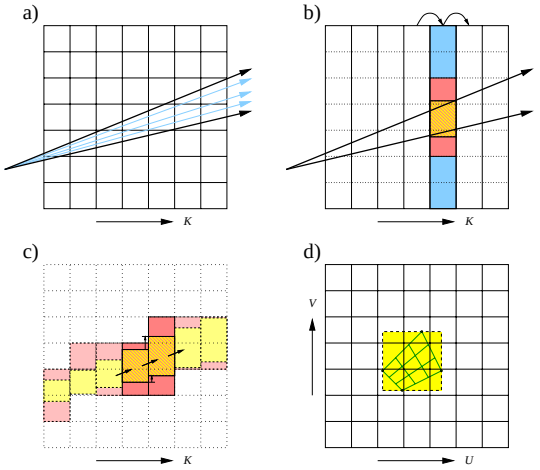

Figure 10: *Given a set of coherent rays, the coherent grid traversal first computes the packet's bounding frustum (a) that is then traversed through the grid one slice at a time (b). For each slice (blue), the frustum's overlap with the slice (yellow) is incrementally computed, which determines the actual cells (red) overlapped by the frustum. (c) Independent of packet size, each frustum traversal step requires only one four-float SIMD addition to incrementally compute the min and max coordinates of the frustum slice overlap, plus one SIMD float-to-int truncation to compute the overlapped grid cells. (d) Viewed down the major traversal axis, each ray packet (green) will have corner rays that define the frustum boundaries (dashed). At each slice, this frustum covers all of the cells covered by the rays.*

the minimum and maximum coordinates of two successive boxes $\mathcal{B}^{(i)}$ and $\mathcal{B}^{(i+1)}$ will differ by a constant vector $\Delta\mathcal{B}$. With each slice being 1 unit wide, this $\Delta\mathcal{B}$ is simply $\Delta\mathcal{B} = (du_{min}, du_{max}, dv_{min}, dv_{max})$, where the $du_{min/max}$ and $dv_{min/max}$ are the slopes of the bounding planes in the grid coordinate space.

Given the overlap box $B^{(i)}$, the next slice's overlap box $B^{(i+1)}$ is incrementally computed via $B^{(i+1)} = B^{(i)} + \Delta B$. This requires only four floating point additions, and can be performed with a single SIMD instruction. As mentioned above, once a slice's overlap box B is known, the range $[i_0..i_1] \times [j_0..j_1]$ of overlapped cells can be determined by truncating $\mathcal{B}$'s coordinates and converting them to integer values. This operation can also be performed with a single SIMD float-to-int conversion instruction. Thus, for arbitrarily sized packets of $N \times N$ rays, the whole process of computing the next slice's overlapped cell coordinates costs only two instructions: one SIMD addition, and one SIMD float-to-int conversion. The complete algorithm is sketched in Figure 10.

7.3. Efficient slice and triangle intersection

Once the cells overlapped by the frustum have been determined, all the rays in a packet are interescted with the triangles in each cell. Triangles may appear in more than one cell, and some rays will traverse cells that would not have been traversed without packets. Consequently, redundant triangle intersection tests are performed. The overhead of these additional tests can be avoided using two well-known techniques: SIMD frustum culling and mailboxing.

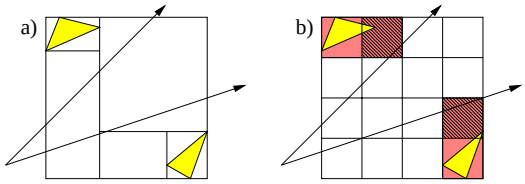

Figure 11: *Since a grid (b) does not adapt as well to the scene geometry as a kd-tree (a), a grid will often intersect triangles (red) that a kd-tree would have avoided. These triangles however usually lie far outside the view frustum, and can be inexpensively discarded by inverse frustum culling during frustum-triangle intersection.*

SIMD frustum culling. A grid does not conform as tightly to the geometry as a kd-tree, and thus requires some triangle intersections that a kd-tree would avoid (see Figure 11). To allow for interactive grid builds, cells are filled if they contain the bounding boxes of triangles rather than the triangles themselves, further exacerbating this problem. However, as one can see in Figure 11, many of these triangles will lie completely outside the frustum; had they intersected the frustum, the kd-tree would have had to perform an intersection test on them as well.

For a packet tracer, triangles outside the bounding frustum can be rejected quite cheaply using Dmitriev et al.'s "SIMD shaft culling" [DHS04]. If the four bounding rays of the frustum miss the triangle on the *same* edge of the triangle, then all the rays must miss that triangle. Using the SIMD triangle intersection method outlined in [Wal04], intersecting the four corner rays costs roughly as much as a single SIMD 4-ray-triangle intersection test. As such, for an N-ray packet, triangles outside the frustum can be intersected at $\frac{4}{N}$ the cost of those inside the frustum.

Mailboxing. In a grid, large triangles may overlap many cells. In addition, since a single-level grid cannot adapt to the position of a triangle, even small triangles often straddle cell boundaries. Thus, most triangles will be referenced in multiple cells. Since these references will be in neighboring cells, there is a high probability that the frustum will intersect the same triangle multiple times. In fact, as shown in Figure 12 this is much more likely for frustum traversal than for a single-ray traversal: While a single ray would visit the same triangle only along one dimension, the frustum is several cells wide, and will re-visit the same triangle in all three dimensions.

Repeatedly intersecting the same triangle can be avoided by mailboxing [KA91]. Each packet is assigned a unique

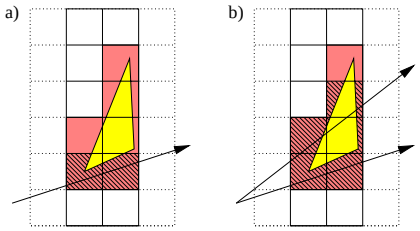

Figure 12: *While one ray (a) can re-visit a triangle in multiple cells only along one dimension, a frustum (b) visits the same triangle much more often (even worse in 3D). These redundant intersection tests would be costly, but can easily be avoided by mailboxing.*

ID, and a triangle is tagged with that ID before the intersection test. Thus, if a packet visits a triangle already tagged with its ID, it can skip intersection. Mailboxing typically produces minimal performance improvements for single ray grids and other acceleration structures, when used for inexpensive primitive such as triangles [Hav02]. As explained above, however, the frustum grid traversal yields far more redundant intersection tests than other acceleration structures and thus profits better from mailboxing. Additionally, the overhead of mailboxing for a packet traverser becomes insignificant; the mailbox test is performed *per packet* instead of per ray, thus amortizing the cost as we have seen before.

Mailboxing and frustum culling are both very useful in reducing the number of redundant intersection tests, and together they remedy the deficiencies of frustum traversal on uniform grids. Ultimately, these two tests will often reduce the number of redundant intersection tests for a frustum grid traversal by an order of magnitude, so that the resulting number of actual ray-triangle intersections tests performed roughly matches that of a kd-tree [WIK*06].

7.4. Extension to hierarchical grids

Wald et al. [WIK*06] show that it is simple to extend the frustum grid traversal to use a multi-level hierarchical grid based on macrocells, and that this noticeably improves performance. Macrocells are a simple hierarchical optimization to a base uniform grid, often used to apply grids to scalar volume fields [PPL*99]. Macrocells superimpose a second, coarser grid over the original fine grid, such that each macrocell corresponds to an $M \times M \times M$ block of original grid cells. Each macrocell stores a Boolean flag specifying whether any of its corresponding grid cells are occupied. Frustum grid traversal with macrocells is simple: the macrocell grid in essence is just an $M \times M \times M$ downscaled version of the original grid, and many of the values computed in the frustum setup can be re-used, or computed by dividing by M. During traversal, one first considers a slice of macrocells, and determine all the macrocells overlapped by the frustum (usually but one in practice). If the macrocells in the slice are all empty, M traversal steps are skipped on the original fine grid. Otherwise, these steps are performed as usual.

Using macrocells was found to improve performance by around 30% [WIK*06], which is consistent with improvements seen for single ray grids. Additional levels of macrocells could further improve performance for more complex models with larger grids, or for zoomed in camera views. More robust varieties of hierarchical grids could speed up large scenes with varying geometric density, at the cost of higher build time.

8. Summary and conclusion

In this STAR, we first discussed the basic design decisions that have to be addressed in the context of real-time ray tracing of dynamic scenes. We concluded that the conflicting goals of real-time traversal performance and per-frame data structure updates/rebuilds add a new dimension to the problem that further complicates the different trade-offs to be taken in any real-time ray tracing system. These trade-offs eventually force us to re-investigate the merits of various data structures, as well as the algorithms used to build them.

We also discussed the most popular acceleration structures – grids, kd-trees, and BVHs – and their respective properties and trade-offs with respect to these design issues, and have covered the various different systems and algorithms proposed for either particular sub-problems (like, for example, how to quickly build a kd-tree), or for complete systems (like Razor, or the respective systems by researchers at Utah, Intel, Saarbrücken, or UNC).

Based on these approaches, we briefly re-visit some of the design questions posed in Section 3. Overall, we still cannot give definitive, conclusive answers to any of these questions. One reason for this is that even though a large number of approaches have been proposed, it is very challenging to compare them to each other because they use different code bases, hardware, optimization levels, traversal algorithms, kinds of motion, test scenes, and ray distributions. Second, with so many factors influencing the relative pros and cons of the individual approaches, the "best" approach will always depend on the actual problem, with some approaches best in some situations, and others in other situations.

Nevertheless, we would at least like to comment on a few issues on which there is a broad consensus at least among the authors of this STAR (which after all represent widely varying schools of thought within the ray tracing community). Because different systems have hard-to-compare performance, it is hard to know which acceleration structure is the fastest, and to what extent performance is likely to change as hardware evolves. Grids are very useful for certain types of scenes and are very fast to build. In particular, for certain dominantly non-axis-aligned scenes if the grid is built with a triangle-in-box test [AM01] it will normally outperform kd-trees and BVHs built using axis aligned splits and bounding boxes. In fact, for certain scenes even a single ray grid will outperform other axis-aligned structures. However, for geometrically wide ray packets grids do not

perform as well. Kd-trees have the fastest reported times for viewing and shadow rays, but they are not easy to update, and it is not clear how well they perform for wide ray packets. Traversal for BVHs is almost as fast as kd-trees, and they can be updated rapidly, but BVHs are about as expensive as kd-trees to rebuild from scratch. Overall, the community should continue to investigate all three approaches, as well as looking onto other possibilities such as oriented structures. The difficulty in actually implementing these different traversal methods is also important to consider. The coherent grid traversal algorithm is likely the most difficult to efficiently implement, while the BVH is the easiest of the three acceleration structures to implement, and would be efficient even without using SIMD instructions. A BIH style BVH is faster to build than a SAH-style BVH, not significantly slower to traverse, and much easier to implement, and therefore is recommended as the first type of build method to implement in an interactive ray tracer.

Concerning whether to rebuild from scratch or rely on updating, the authors agree that future systems will likely use a combination of both, where rebuilding from scratch every frame is used some of the time and/or for some parts of the hierarchy, and refitting or incremental updates are used for the deformable parts of the scene when it does not introduce too much degradation. Lazy or partial builds are likely to receive more attention, but require active support from the application. This argues for some co-existence of both approaches depending on whether the application provides such information, or whether it only produces a "direct rendering mode" triangle soup; the same is true for hierarchical techniques and multiresolution approaches.

These statements reflect the authors' personal opinions, and future research may change some of these conclusions.

In general, adding animated scenes to real-time ray tracing has made ray tracing research considerably more varied; and more interesting, too, by having opened new questions, and by having re-opened old ones that had already been considered solved. Though the field has recently seen tremendous progress, there is no clear winner, yet, and arguably, with so many different variations of the problems no single technique can ever be best in all cases.

Despite the flurry of recently published systems, the space of as yet unexplored combinations is still huge. In particular, future work is likely to focus on better evaluating the relative strengths and weaknesses of kd-trees and BVH: for example, fast, approximate, and scalable parallel builds are known for kd-trees, and should apply similarly to BVHs, but have not been fully investigated, yet; the same is true for the various BVH-based respectively kd-tree-based traversal algorithms. How these approaches compare with respect to different hardware architectures like GPUs or upcoming multi-core architectures is also interesting, as is the question how to handle wider than four SIMD widths, or more general secondary ray packets. Multiresolution geometry and lazy/partial builds require more attention, but ultimately ray tracers have to be integrated into real-world graphics workloads to see how these approaches behave (the same is true for triangle soup approaches). Finally, *all* of the systems discussed above depend on packets and frustum techniques to achieve high performance, but apart from the obvious question on how different ray distributions work for the various data structures (that we partially addressed above), the more general question of how to use these techniques in a "real" rendering system (i.e., where these packets come from in the first place) is an open question for future research.

Acknowledgments

We are grateful to a large number of people that have provided feedback and/or insight into their respective papers and systems. In particular, Sung-Eui Yoon and Christian Lauterbach have provided feedback on their BVH-based systems, and, in particular, on selective restructuring. Vlastimil Havran has provided invaluable feedback on skd-tree like data structures and on the relation of the different data structures in general. Virtually all of the images and performance numbers have been selected from other people's systems and papers; though all of these are cited, these papers contain additional acknowledgments that we have omitted.

With few exceptions, this work surveys existing results, and draws from existing publications that others have significantly contributed to: For Razor, we explicitly acknowledge Gordon Stoll, Peter Djeu, Don Fussell, Ikrima Elhassan, Rui Wang, and Denis Zorin; for the Grid section, Aaron Knoll and Andrew Kensler; and for BVHs and kd-trees, Heiko Friedrich, Carsten Benthin, and Philipp Slusallek.

The writing of this survey has been supported by the National Science Foundation (awards #0541009, #0306151, and CAREER award #0546236), by the U.S. Department of Energy through the Center for the Simulation of Accidental Fires and Explosions (grant W-7405-ENG-48LA-13111-PR), and by research grants from Intel Corporation. The authors would particularly like to thank Jim Hurley at Intel, who has strongly supported academic ray tracing research over the past several years.

References

[AGM06] ABERT O., GEIMER M., MÜLLER S.: Direct and Fast Ray Tracing of NURBS Surfaces. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 161–168.

[AM01] AKENINE-MÖLLER T.: Fast 3D triangle-box overlap testing. *Journal of Graphics Tools 6*, 1 (2001), 29–33.

[BEL*07] BOULOS S., EDWARDS D., LACEWELL J. D., KNISS J., KAUTZ J., SHIRLEY P., WALD I.: Packet-based Whitted and Distribution Ray Tracing. In *Proceedings of Graphics Interface 2007* (May 2007).

[Ben06] BENTHIN C.: *Realtime Ray Tracing on current CPU Architectures*. PhD thesis, Saarland University, 2006.

[BGES98] BARTZ D., GROSSO R., ERTL T., STRAER W.: Parallel construction and isosurface extraction of recursive tree structures. In *Proceedings of WSCG'98* (1998), vol. 3.

[BWS04] BENTHIN C., WALD I., SLUSALLEK P.: Interactive Ray Tracing of Free-Form Surfaces. In *Proceedings of Afrigraph* (November 2004), pp. 99–106.

[BWS06] BOULOS S., WALD I., SHIRLEY P.: *Geometric and Arithmetic Culling Methods for Entire Ray Packets*. Tech. Rep. UUCS-06-010, SCI Institute, University of Utah, 2006.

[BWSF06] BENTHIN C., WALD I., SCHERBAUM M., FRIEDRICH H.: Ray Tracing on the CELL Processor. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 15–23.

[CFLB06] CHRISTENSEN P. H., FONG J., LAUR D. M., BATALI D.: Ray tracing for the movie 'Cars'. In *Proc. IEEE Symposium on Interactive Ray Tracing* (2006), pp. 1–6.

[CLF*03] CHRISTENSEN P. H., LAUR D. M., FONG J., WOOTEN W. L., BATALI D.: Ray Differentials and Multiresolution Geometry Caching for Distribution Ray Tracing in Complex Scenes. In *Computer Graphics Forum (Eurographics 2003 Conference Proceedings)* (September 2003), Blackwell Publishers, pp. 543–552.

[CPC84] COOK R., PORTER T., CARPENTER L.: Distributed Ray Tracing. *Computer Graphics (Proceeding of SIGGRAPH 84) 18*, 3 (1984), 137–144.

[DFG99] DU Q., FABER V., GUNZBURGER M.: Centroidal voronoi tessellations: Applications and algorithms. *SIAM Rev. 41*, 4 (1999), 637–676.

[DHS04] DMITRIEV K., HAVRAN V., SEIDEL H.-P.: *Faster Ray Tracing with SIMD Shaft Culling*. Research Report MPI-I-2004-4-006, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 2004.

[DHW*07] DJEU P., HUNT W., WANG R., ELHASSAN I., STOLL G., MARK W. R.: *Razor: An Architecture for Dynamic Multiresolution Ray Tracing*. Tech. rep., University of Texas at Austin Dep. of Comp. Science, 2007. Conditionally accepted to ACM Transactions on Graphics.

[Ebe05] EBERLY D. H.: *3D Game Engine Architecture: Engineering Real-Time Applications with Wild Magic*. Morgan Kaufmann, 2005.

[EGMM06] EISEMANN M., GROSCH T., MAGNOR M., MUELLER S.: *Automatic Creation of Object Hierarchies for Ray Tracing of Dynamic Scenes*. Tech. Rep. 2006-6-1, TU Braunschweig, 2006.

[FGD*06] FRIEDRICH H., GÜNTHER J., DIETRICH A., SCHERBAUM M., SEIDEL H.-P., SLUSALLEK P.: Exploring the use of ray tracing for future games. In *sandbox '06: Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames* (New York, NY, USA, 2006), ACM Press, pp. 41–50.

[GFSS06] GÜNTHER J., FRIEDRICH H., SEIDEL H.-P., SLUSALLEK P.: Interactive ray tracing of skinned animations. *The Visual Computer 22*, 9 (Sept. 2006), 785–792. (Proceedings of Pacific Graphics).

[GFW*06] GÜNTHER J., FRIEDRICH H., WALD I., SEIDEL H.-P., SLUSALLEK P.: Ray tracing animated scenes using motion decomposition. *Computer Graphics Forum 25*, 3 (Sept. 2006), 517–525. (Proceedings of Eurographics).

[GG91] GERSHO A., GRAY R. M.: *Vector quantization and signal compression*. Kluwer Academic Publishers, Norwell, MA, USA, 1991.

[GG98] GAEDE V., GÜNTHER O.: Multidimensional access methods. *ACM Computing Surveys 30*, 2 (1998), 170–231.

[GIK*07] GRIBBLE C. P., IZE T., KENSLER A., WALD I., PARKER S. G.: A coherent grid traversal approach to visualizing particle-based simulation data. *IEEE Transactions on Visualization and Computer Graphics 13*, 4 (2007), 758–768.

[GLM96] GOTTSCHALK S., LIN M., MANOCHA D.: OBB-Tree: a hierarchical structure for rapid interference detection. In *Proceedings of SIGGRAPH 96* (Aug. 1996), Computer Graphics Proceedings, Annual Conference Series, pp. 171–180.

[GS87] GOLDSMITH J., SALMON J.: Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics and Applications 7*, 5 (1987), 14–20.

[Hai91] HAINES E.: Efficiency improvements for hierarchy traversal in ray tracing. In *Graphics Gems II*, Arvo J., (Ed.). Academic Press, 1991, pp. 267–272.

[Hav97] HAVRAN V.: Cache Sensitive Representation for the BSP Tree. In *Compugraphics'97* (December 1997), GRASP – Graphics Science Promotions & Publications, pp. 369–376.

[Hav01] HAVRAN V.: *Heuristic Ray Shooting Algorithms*. PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, 2001.

[Hav02] HAVRAN V.: Mailboxing, Yea or Nay? *Ray Tracing News 15*, 1 (2002).

[Hav07] HAVRAN V.: About the relation between spatial subdivision and object hierarchies used in ray tracing. In *Proceedings of the Spring Conference on Computer Graphics (SCCG)* (2007).

[HEV*04] HADAP S., EBERLE D., VOLINO P., LIN M. C., REDON S., ERICSON C.: Collision detection and proximity queries. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes* (2004), ACM Press, p. 15.

[HH84] HECKBERT P. S., HANRAHAN P.: Beam tracing polygonal objects. In *Proceedings of SIGGRAPH* (1984), pp. 119–127.

[HHS06] HAVRAN V., HERZOG R., SEIDEL H.-P.: On Fast Construction of Spatial Hierarchies for Ray Tracing. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006).

[HMFS07] HUNT W., MARK W. R., FUSSELL D., STOLL G.: Fast construction of ray-tracing acceleration structures. In preparation for submission to 2007 IEEE Symp. on Interactive Ray Tracing, 2007.

[HPP00] HAVRAN V., PRIKRYL J., PURGATHOFER W.: *Statistical comparison of ray-shooting efficiency schemes*. Tech. Rep. TR-186-2-00-14, Institute of Computer Graphics, Vienna University of Technology, 2000.

[HSM06] HUNT W., STOLL G., MARK W.: Fast kd-tree Construction with an Adaptive Error-Bounded Heuristic. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006).

[Ige99] IGEHY H.: Tracing ray differentials. In *Proceedings of SIGGRAPH 99* (Aug. 1999), Computer Graphics Proceedings, Annual Conference Series, pp. 179–186.

[IWP07] IZE T., WALD I., PARKER S. G.: Asynchronous BVH Construction for Ray Tracing Dynamic Scenes on Parallel Multi-Core Architectures. In *Proceedings of the 2007 Eurographics Symposium on Parallel Graphics and Visualization* (2007).

[IWRP06] IZE T., WALD I., ROBERTSON C., PARKER S. G.: An Evaluation of Parallel Grid Construction for Ray Tracing Dynamic Scenes. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 47–55.

[JP04] JAMES D. L., PAI D. K.: BD-Tree: Output-sensitive collision detection for reduced deformable models. *ACM Transactions on Graphics (SIGGRAPH 2004) 23*, 3 (Aug. 2004).

[KA91] KIRK D., ARVO J.: Improved ray tagging for voxel-based ray tracing. In *Graphics Gems II*, Arvo J., (Ed.). Academic Press, 1991, pp. 264–266.

[KH95] KEATES M. J., HUBBOLD R. J.: Interactive ray tracing on a virtual shared-memory parallel computer. *Computer Graphics Forum 14*, 4 (Oct. 1995), 189–202.

[KK86] KAY T., KAJIYA J.: Ray tracing complex scenes. In *Proceedings of SIGGRAPH* (1986), pp. 269–278.

[Knu98] KNUTH D. E.: *The Art of Computer Programming, Volume 3: Sorting and Searching*, second ed. Addison-Wesley, 1998.

[LAM01] LEXT J., AKENINE-MÖLLER T.: Towards Rapid Reconstruction for Animated Ray Tracing. In *Eurographics Short Presentations* (2001), pp. 311–318.

[LAM05] LARSSON T., AKENINE-MÖLLER T.: A dynamic bounding volume hierarchy for generalized collision detection. In *Workshop On Virtual Reality Interaction and Physical Simulation* (2005), pp. 91–100.

[LCF05] LUQUE R. G., COMBA J. L. D., FREITAS C. M. D. S.: Broad-phase collision detection using semi-adjusting bsp-trees. In *I3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games* (2005), ACM Press, pp. 179–186.

[Llo82] LLOYD S. P.: Least Squares Quantization in PCM. *IEEE Transactions on Information Theory 28*, 2 (1982), 129–137.

[LM04] LIN M. C., MANOCHA D.: Collision and proximity queries. In *Handbook of Discrete and Computational Geometry, 2nd Edition*. CRC Press, 2004, pp. 787–807.

[LYTM06] LAUTERBACH C., YOON S.-E., TUFT D., MANOCHA D.: RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 39–45.

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

[MB89] MACDONALD J. D., BOOTH K. S.: Heuristics for Ray Tracing using Space Subdivision. In *Proceedings of Graphics Interface* (1989), pp. 152–63.

[MB90] MACDONALD J. D., BOOTH K. S.: Heuristics for ray tracing using space subdivision. *Visual Computer 6*, 6 (1990), 153–65.

[MCEF94] MOLNAR S., COX M., ELLSWORTH D., FUCHS H.: A sorting classification of parallel rendering. *IEEE Comput. Graph. Appl. 14*, 4 (1994), 23–32.

[MF99] MÜLLER G., FELLNER D.: Hybrid Scene Structuring with Application to Ray Tracing. In *Proceedings of International Conference on Visual Computing* (1999), pp. 19–26.

[MF05] MARK W., FUSSELL D.: *Real-Time Rendering Systems in 2010*. Tech. Rep. 05-18, Computer Science, University of Texas, May 2005.

[ML03] MASSO J. P. M., LOPEZ P. G.: Automatic Hybrid Hierarchy Creation: a Cost-model Based Approach. *Computer Graphics Forum 22*, 11 (2003), 513.

[Muu95] MUUSS M.: Towards real-time ray-tracing of combinatorial solid geometric models. In *Proceedings of BRL-CAD Symposium* (1995).

[MW04] MAHOVSKY J., WYVILL B.: Fast ray-axis aligned bounding box overlap tests with Plücker coordinates. *Journal of Graphics Tools 9*, 1 (2004), 35–46.

[MW06] MAHOVSKY J., WYVILL B.: Memory-Conserving Bounding Volume Hierarchies with Coherent Raytracing. *Computer Graphics Forum 25*, 2 (June 2006).

[ORM07] OVERBECK R., RAMAMOORTHI R., MARK W. R.: A Real-time Beam Tracer with Application to Exact Soft Shadows. In *Proceedings of the Eurographics Symposium on Rendering* (2007).

[OSDM87] OOI B. C., SACKS-DAVIS R., MCDONNELL K. J.: Spatial k-d-tree: An indexing mechanism for spatial databases. In *IEEE International Computer Software and Applications Conference (COMPSAC)* (1987).

[PGSS06] POPOV S., GÜNTHER J., SEIDEL H.-P., SLUSALLEK P.: Experiences with Streaming Construction of SAH KD-Trees. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006).

[PH04] PHARR M., HUMPHREYS G.: *Physically Based Rendering : From Theory to Implementation*. Morgan Kaufman, 2004.

[PMS*99] PARKER S. G., MARTIN W., SLOAN P.-P. J., SHIRLEY P., SMITS B. E., HANSEN C. D.: Interactive ray tracing. In *Proceedings of Interactive 3D Graphics* (1999), pp. 119–126.

[PPL*99] PARKER S., PARKER M., LIVNAT Y., SLOAN P.-P., HANSEN C., SHIRLEY P.: Interactive ray tracing for volume visualization. *IEEE Trans. on Computer Graphics and Visualization 5*, 3 (1999), 238–250.

[PSL*98] PARKER S., SHIRLEY P., LIVNAT Y., HANSEN C., SLOAN P.-P.: Interactive Ray Tracing for Isosurface Rendering. In *IEEE Visualization '98* (October 1998), pp. 233–238.

[Res06] RESHETOV A.: Omnidirectional ray tracing traversal algorithm for kd-trees. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 57–60.

[RSH00] REINHARD E., SMITS B., HANSEN C.: Dynamic acceleration structures for interactive ray tracing. In *Proceedings of the Eurographics Workshop on Rendering* (Brno, Czech Republic, June 2000), pp. 299–306.

[RSH05] RESHETOV A., SOUPIKOV A., HURLEY J.: Multi-Level Ray Tracing Algorithm. *ACM Transaction*

on Graphics 24, 3 (2005), 1176–1185. (Proceedings of ACM SIGGRAPH 2005).

[RW80] RUBIN S., WHITTED T.: A 3D representation for fast rendering of complex scenes. In *Proceedings of SIGGRAPH* (1980), pp. 110–116.

[Sam89a] SAMET H.: *Applications of Spatial Data Structures: Computer Graphics, Image Processing and GIS*. Addison-Wesley, 1989.

[Sam89b] SAMET H.: *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1989.

[Sam06] SAMET H.: *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.

[San02] SANTALO L.: *Integral Geometry and Geometric Probability*. Cambridge University Press, 2002.

[SDP*04] SCHMITTLER J., DAHMEN T., POHL D., VOGELGESANG C., SLUSALLEK P.: Ray Tracing for Current and Future Games. In *Proceedings of 34. Jahrestagung der Gesellschaft für Informatik* (2004).

[Sed98] SEDGEWICK R.: *Algorithms in C++, Parts 1-4: Fundamentals, Data Structure, Sorting, Searching*. Addison Wesley, 1998. (3rd Ed.).

[Smi98] SMITS B.: Efficiency issues for ray tracing. *Journal of Graphics Tools 3*, 2 (1998), 1–14.

[SSK07] SHEVTSOV M., SOUPIKOV A., KAPUSTIN A.: Fast and scalable kd-tree construction for interactively ray tracing dynamic scenes. *Computer Graphics Forum 26*, 3 (2007). (Proceedings of Eurographics), to appear.

[Sto05] STOLL G.: Part II: Achieving Real Time / Optimization Techniques. Slides from the Siggraph 2005 Course on Interactive Ray Tracing, 2005. Slides available online at http://www.openrt.de/Siggraph05/-UpdatedCourseNotes/course.php.

[Sub90] SUBRAMANIAN K. R.: *A Search Structure based on K-d Trees for Efficient Ray Tracing*. PhD thesis, The University of Texas at Austin, Dec. 1990.

[SW01] SUYKENS F., WILLEMS Y.: Path differentials and applications. In *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering* (June 2001), pp. 257–268.

[SWS02] SCHMITTLER J., WALD I., SLUSALLEK P.: SaarCOR – A Hardware Architecture for Ray Tracing. In *Proceedings of the ACM SIGGRAPH/Eurographics Conference on Graphics Hardware* (2002), pp. 27–36.

[TKH*05] TESCHNER M., KIMMERLE S., HEIDELBERGER B., ZACHMANN G., RAGHUPATHI L., FUHRMANN A., CANI M., FAURE F., MAGNENAT-THALMANN N., STRASSER W., VOLINO P.: Collision detection for deformable objects. *Computer Graphics Forum 24*, 1 (2005), 61–82.

[TL04] TABELLION E., LAMORLETTE A.: An approximate global illumination system for computer generated films. In *Proceedings of SIGGRAPH* (2004), pp. 469–476.

[vdB97] VAN DEN BERGEN G.: Efficient collision detection of complex deformable models using AABB trees. *Journal of Graphics Tools 2*, 4 (1997), 1–14.

[Wal04] WALD I.: *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Saarland University, 2004.

[WBMS05] WILLIAMS A., BARRUS S., MORLEY R. K., SHIRLEY P.: An efficient and robust ray-box intersection algorithm. *Journal of Graphics Tools 10*, 1 (2005), 49–54.

[WBS02] WALD I., BENTHIN C., SLUSALLEK P.: *OpenRT - A Flexible and Scalable Rendering Engine for Interactive 3D Graphics*. Tech. rep., Saarland University, 2002. Available at http://graphics.cs.uni-sb.de/Publications.

[WBS03] WALD I., BENTHIN C., SLUSALLEK P.: Distributed Interactive Ray Tracing of Dynamic Scenes. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics* (2003), pp. 11–20.

[WBS07] WALD I., BOULOS S., SHIRLEY P.: Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics 26*, 1 (2007), 1–18.

[WFKH07] WALD I., FRIEDRICH H., KNOLL A., HANSEN C. D.: *Interactive Isosurface Ray Tracing of Time-Varying Tetrahedral Volumes*. Tech. Rep. UUSCI-2007-003, SCI Institute, University of Utah, 2007. (conditionally accepted at IEEE Visualization 2007).

[WH06] WALD I., HAVRAN V.: 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* (2006), pp. 61–70.

[Whi80] WHITTED T.: An Improved Illumination Model for Shaded Display. *Communications of the ACM 23*, 6 (1980), 343–349.

[WIK*06] WALD I., IZE T., KENSLER A., KNOLL A., PARKER S. G.: Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM Transactions on Graphics 25*, 3 (2006), 485–493. (Proceedings of ACM SIGGRAPH).

[WK06] WÄCHTER C., KELLER A.: Instant Ray Tracing: The Bounding Interval Hierarchy. In *Rendering Techniques 2006 – Proceedings of the 17th Eurographics Symposium on Rendering* (2006), pp. 139–149.

[WMS06] WOOP S., MARMITT G., SLUSALLEK P.: B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes. In *Proceedings of Graphics Hardware* (2006).

[WSBW01] WALD I., SLUSALLEK P., BENTHIN C., WAGNER M.: Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum 20*, 3 (2001), 153–164. (Proceedings of Eurographics).

[WSS05] WOOP S., SCHMITTLER J., SLUSALLEK P.: RPU: A programmable ray processing unit for realtime ray tracing. *ACM Transactions on Graphics 24*, 3 (2005), 434–444. (Proceedings of SIGGRAPH).

[YCM07] YOON S.-E., CURTIS S., MANOCHA D.: Ray Tracing Dynamic Scenes using Selective Restructuring. In *Eurographics Symposium on Rendering* (2007).

[YM06] YOON S., MANOCHA D.: Cache-Efficient Layouts of Bounding Volume Hierarchies. *Computer Graphics Forum (Eurographics)* (2006).