Copyright
by
Warren Andrew Hunt
2008

The Dissertation Committee for Warren Andrew Hunt
certifies that this is the approved version of the following dissertation:

Data Structures and Algorithms for Real-Time Ray Tracing at the University of Texas at Austin

Committee:

William Mark, Supervisor

Donald Fussell

Okan Arikan

Gregory Plaxton

Peter Shirley

Data Structures and Algorithms for Real-Time Ray Tracing at the University of Texas at Austin

by

Warren Andrew Hunt, B.S., B.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2008

Dedicated to the real-time ray tracing community.

Acknowledgments

I wish to thank the multitudes of people who have helped me along the way. In particular, I would like to thank my lab-mates: Peter Djeu, Sean Keeley, Gilbert Bernstein and Paul Navratil for their discussions and feedback on the topics of geometry and ray tracing and Jim Hurley and Intel for supporting my work with encouragement, a job and a fellowship.

Data Structures and Algorithms for Real-Time Ray Tracing at the University of Texas at Austin

Publication No. _____

Warren Andrew Hunt, Ph.D.
The University of Texas at Austin, 2008

Supervisor: William Mark

Modern rendering systems require fast and efficient acceleration structures in order to compute visibility in real time. I present several novel data structures and algorithms for computing visibility with high performance. In particular, I present two algorithms for improving heuristic based acceleration structure build. These algorithms, when used in a demand driven way, have been shown to improve build performance by up to two orders of magnitude. Additionally, I introduce ray tracing in perspective transformed space. I demonstrate that ray tracing in this space can significantly improve visibility performance for near-common origin rays such as eye and shadow rays. I use these data structures and algorithms to support a key hypothesis of this dissertation: "There is no silver bullet for solving the visibility problem; many different acceleration structures will be required to achieve the highest perfor-

mance." Specialized acceleration structures provide significantly better performance than generic ones and building many specialized structures requires high performance build techniques. Additionally, I present an optimization-based taxonomy for classifying acceleration structures and algorithms in order to identify which optimizations provide the largest improvement in performance. This taxonomy also provides context for the algorithms I present. Finally, I present several novel cost metrics (and a correction to an existing cost metric) to improve visibility performance when using metric based acceleration structures.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Virtual environments have greatly changed the face of many fields such as entertainment and medicine over the past 20 years. The most stimulating/meaningful interactions with such environments are visual. Visual interactions with virtual environments can both inform and entertain. The field of computer graphics studies ways to enhance such virtual environments and their visual interface. Rendering is the study of producing visual output from the description of a virtual environment. My research is dedicated to the study of efficient methods of computing a particular piece of the rendering process known as visibility.

Visibility refers to two related problems, stated as follows:

1. "Given a scene (virtual environment), a point in space and a direction (a ray), what scene object does the ray intersect first?"

2. "Given a scene and two points, is any scene object directly between the two points?"

The second query is less general and can be phrased in terms of the first. However, it also requests less information, asking "if" rather than "what".

This distinction has important implications in terms of performance. These queries can be integrated to form more complicated queries such as:

"Given a scene, a point and a solid angle, what are all of the scene objects intersected by some ray originating at the point with some angle from the given solid angle?"

Millions of such queries are required to compute a single image of a virtual environment for a modern game, medical visualization or simulation. If real-time frame-rates are required for immersive play or real-time feedback, potentially hundreds of millions or even billions of such queries will be required per second. Due to the sheer number of these operations, computing them efficiently is extremely important for real-time performance. Many optimizations are employed in every modern rendering system to make these queries more efficient. I present several new acceleration structures and algorithms for more efficiently solving specific cases of the visibility problem. In this context, acceleration structures are organizational structures that reduce the number of ray-object intersection tests. Rays traverse these structures in order to find objects they are likely to intersect and avoid objects they are unlikely to intersect. Ray tracing and acceleration structures are covered in the familiarization section of this chapter.

Efficient acceleration structures and rendering algorithms are required to achieve real-time performance using a ray tracer. However, these data structures shouldn't add a prohibitive amount of (traversal) overhead. Acceleration structures must be fast to build in order to support dynamic virtual

environments (where the structure needs to change between frames).

Classically, ray tracing was an offline procedure and acceleration structure build was also an offline procedure. Given modern hardware and algorithms ray tracing can now be performed in real time [58]. However, acceleration structure build (circa 2006) cannot and thus ray tracing cannot support dynamic environments. Faster building of acceleration structures for dynamic scenes was unanimously voted to be the most important problem at the interactive ray tracing course at Siggraph '05. In response, I present several algorithms for improving build times for traditional acceleration structures for ray tracing. Combinations of these improvements have been shown to improve build performance by as much as two orders of magnitude in certain cases.

Because performance is often the central theme of real-time rendering, I additionally present a systematic evaluation of many of the optimizations commonly used in rendering systems in order to discuss their relative merits and divine which are most effective. In order to do so, I present a taxonomy for classifying acceleration structures and rendering algorithms according to their optimizations. Using the taxonomy I show explicitly that z-buffer rendering (the high-performance algorithm commonly used in games and implemented in hardware by nVidia, ATI and others) is a special case of ray tracing and I provide an "algorithmic pathway" between traditional z-buffering and traditional ray tracing. This path highlights novel acceleration structures and rendering algorithms with many of the properties of both ray tracing and z-buffering. In particular, this taxonomy highlights the perspective transform as

the primary constituent in the high performance of the z-buffer algorithm. I demonstrate acceleration structures for ray tracing primary visibility and true shadows with performance comparable to z-buffering. These structures also have the potential to produce real-time soft-shadow, depth of field and motion blur effects. I provide results for a software system which achieves real-time frame rates for primary visibility and hard shadows on a modern CPU at high resolution.

In order to better understand the performance of acceleration structures, I discuss cost metrics used to build "good" acceleration structures. I present several new cost metrics that make better assumptions about the process they are accelerating. I also discuss cost metrics that take into account different hardware features such as memory usage. Lastly, I present a correction to the most commonly used cost metric (the surface area metric) in order to take into account the effects of another common optimization (mailboxing).

1.1 Working Hypotheses

Most of my research is premised on the following assumptions:

1. There is no silver bullet for real-time rendering. Visibility queries are diverse in both assumptions and results.

2. Visibility queries can be classified into groups according to properties of those queries. The highest performance for a given group of queries

will be obtained by an acceleration structure and rendering algorithm specifically tailored to that group.

3. The reduction in traversal and intersection costs using specialized acceleration structures outweighs the additional cost of building those specialized structures. (I demonstrate this to be true a majority of the time, even without advanced fast-build techniques.)

I hypothesize that the combination of many specialized acceleration structures and rendering algorithms into a single system will provide the highest overall performance. I cannot divine the ultimate rendering system, but I provide evidence to support this hypothesis in this dissertation.

1.2 Familiarization

Before discussing my contributions to the field of real-time ray tracing, I'm going to take a short diversion to provide some background for an introduction to the field itself.

1.2.1 Recursive Ray Tracing

Recursive ray tracing was introduced around 1980 by Kay [46] and Turner Whitted [75]. The algorithm roughly reflects the reality of light transport: rays are bounced around a scene in a similar manner to the way light bounces around in the real world. Whitted's ray tracing algorithm launched one ray from the origin of a virtual pinhole camera for each pixel in the output

image. The directions of the rays are determined by the locations of the pixels on the image plane. When these camera rays intersect objects in the scene they interact with those objects. Rays intersecting diffuse objects obtain a color for the pixel they corresponded to. Rays intersecting reflective or refractive objects spawn new rays traveling in the directions determined by the laws governing reflection and refraction. These reflective and refractive rays may strike diffuse objects and obtain a color or recursively interact with other reflective or refractive objects. This simple recursive algorithm introduced the world to ray tracing.

1.2.2 Distribution Ray Tracing

In 1984, Cook introduced distribution ray tracing [18]. Distribution ray tracing is a stochastic algorithm that uses multiple rays to enable the rendering of "fuzzy" effects. These effects include soft shadows caused by area lights, depth of field caused by non idealized cameras, motion blur and glossy reflections and refractions. Distribution ray tracing uses multiple rays to integrate visibility queries over an area or time using a Monte-Carlo approach. For example, the partial coverage of an area light may be determined by sampling a stochastically selected set of points on the surface of the light (each with a ray). Distribution ray tracing is capable of producing substantially more convincing images and is the basis of current ray tracing technology used in offline applications. In addition to producing extremely high quality images, distribution ray tracing is known for being extremely expensive. In practice,

a high resolution distribution ray traced image can use many billions of rays.

1.2.3 Acceleration Structures

Although these ray tracing algorithms are capable of producing very convincing images, they cannot do so quickly. If every ray is tested for intersection against every object in a scene, the amount of work required to render a scene with large amount of geometry is prohibitively high. To avoid performing so many intersection tests, ray tracers employ acceleration structures. One of the earliest acceleration structures is the uniform grid [16]. Using a grid, objects are sorted into the spatial cells as a pre-processing step. Rays then step through the grid cells and only intersect geometry found in those cells. Other acceleration structures include binary space partitioning [22] (BSP) trees (including kd-trees [9] and oct-trees [42]) which recursively partition space and bounding volume hierarchies [25, 61] (BVHs) which recursively aggregate scene objects.

All acceleration structures can significantly reduce the number of ray-object intersection tests performed per ray. However, these structures add their own overhead. Acceleration structures must be constructed and traversed. Much of high-performance ray tracing research is dedicated to finding "good" acceleration structures (that effectively reduce the number of intersection tests) and building and traversing those structures quickly.

1.3 Overview of Contribution

The contributions in this dissertation relate to data structures and algorithms for high-performance ray tracing. The contributions include novel acceleration structure build algorithms, novel applications of spatial transforms to improve rendering performance and new and corrected cost metrics to improve the effectiveness of acceleration structures. I use these data structures and algorithms to support my working hypothesis from the previous section.

In addition to supporting these hypotheses, I intend for a primary contribution of this dissertation to be a conceptual framework for classifying acceleration structures and rendering algorithms. This framework will be used to justify (or condemn) optimizations commonly used in modern rendering systems. I hope that this taxonomy and discussion provides the community a better understanding of the science of building such structures and algorithms and why these systems behave the way they do, leading to more principled designs in the future. In addition, the taxonomy illuminates a continuum of algorithms between ray tracing and z-buffering. I intend for this continuum to help generate better high-performance ray tracing structures and more seamlessly integrate rasterization and ray tracing.

I present two novel algorithms for improving the performance of heuristic based kd-tree build. Heuristics are commonly employed during the kd-tree build process. For any particular scene, an extremely large number of valid acceleration structures exist. A build algorithm uses heuristics to choose an

acceleration structure that is expected to provide high traversal performance. These heuristics help choose scene partitions that more effectively reduce the number of ray-object intersection tests performed per ray but add overhead in the process. One optimization uses sampling and reconstruction to closely approximate the standard heuristic (a cost function). The other uses additional (commonly available) outside information to reduce build times from $O(n \log n)$ to $O(n)$. These two algorithms, used in conjunction in a demand driven way, are shown to yield up to two orders of magnitude speed improvement for kd-tree build over large scenes. In this case demand driven refers to the acceleration structure build. In the system I describe, the acceleration structure is only constructed as needed.

Additionally, I introduce methods for using the perspective transform to accelerate ray tracing for collections of rays with common or nearly-common origin. Using acceleration structures under the perspective transform, I demonstrate primary and hard-shadow performance many times faster than other dynamic ray tracing techniques and even competitive with a software implementation of the z-buffer algorithm. These structures are key to my argument that specialized acceleration structures will achieve the highest performance for ray tracing.

Finally, I provide a discussion with regard to the nature of the visibility problem and how to use cost metrics to capture this nature. I present several new cost metrics, including one that makes substantially more accurate assumptions under the perspective transform. I also provide a correction to

the most commonly used cost metric to account for a specific optimization (mailboxing) that changes its character.

Chapter 2

Background and Related Work

Acceleration of visibility queries is a very well studied problem [20]. To make sense of the plethora of solutions that exist, it is important to understand the visibility problem at a high level. A simple way to view the visibility problem is as a cross product (or database "join") between two collections: one collection (of size m) of samples (rays, beams etc.) and one collection (of size n) of scene objects (triangles, implicit surfaces, etc.). It should be noted that this cross product is more general than required in practice. Recall that visibilities are split into two categories. Visibility queries of the first kind ("Given a scene, a point in space and a direction, what scene object does the ray intersect first?") only require the closest intersection test and visibility queries of the second kind only require the existence of an intersection. Because of these particulars, the visibility problem doesn't *exactly* resemble a "join".

Without an acceleration structure the visibility join consists of $m \times n$ operations. The use of an acceleration structure culls many non-intersecting pairs of the join. For queries of the first kind all sample-object pairs that don't result in the closest intersection may be culled without changing the result of the query. This culling can substantially improve the performance of the join.

If accelerated perfectly each sample (e.g., ray) is only tested for intersection against at most one scene object (resulting in $\leq m$ intersections). In practice perfect acceleration is almost impossible to achieve but $O(m)$ intersections is common.

As mentioned in the familiarization section, the most common acceleration structures are grids, BSP-trees and BVHs. These structures fall into two categories: those that use aggregation and those that use partitioning.

Aggregation based acceleration structures group samples or objects into bundles [61]. These aggregations are conservatively tested for intersection with each other in the hopes that they do not intersect. If they do not, no element in either aggregate could have intersected any element in the other aggregate. The most traditional aggregation based acceleration structure is a bounding volume hierarchy (BVH). Using a BVH, individual rays (samples) are tested against geometry aggregates (represented by bounding volumes).

Partitioning-based acceleration structures partition space (not necessarily 3-space) into non-overlapping regions using surfaces (usually axis-aligned hyper planes)[16, 22]. Only samples and objects that exist in the same partition may intersect. The classic partitioning schemes are grids and BSP trees (including kd-trees and oct-trees). Spatial partitioning structures are broken into two categories: uniform and hierarchical. Uniform structures implicitly divide space into a collection of partitions (often grid-cells). Hierarchical structures recursively subdivide space. Hierarchical structures are more general than uniform structures. Any spatial partition defined by a uniform scheme

can be defined by a hierarchical one. Uniform spatial structures, on the other hand, have an implicitly defined topology that makes their creation and management much simpler. Between the two types there exists a trade-off between generality and simplicity.

Aggregation and partitioning structures have notably different properties in some respects. The most noticeable difference is the type of redundancy inherent to the structure. Partitioning structures have the property that each point in space exists in exactly one partition but admit scene objects to occur in multiple partitions. Object aggregation hierarchies, on the other, hand have the property that each scene object exists at exactly one point in the hierarchy, but any given point in space could occur within multiple points in the hierarchy.

Although once a point of contention amongst the community, it is now clear that neither type of acceleration structure is strictly superior to the other. I will discuss the technical merits of each type in a later section. It should be noted that even though aggregation schemes typically focus on object hierarchies, some research has been done into sample aggregation [59].

2.1 Cost Metrics Background

Many different acceleration structures of each type exist for any given scene. These different structures, however, are not equally effective. To evaluate the effectiveness of a particular structure, researchers have developed cost metrics for acceleration structures. These metrics traditionally attempt to pre-

dict the number of intersections performed during ray tracing. Cost metrics may be used as part of a heuristic for improving acceleration structure quality during the build process.

The most commonly used cost metric is the surface area metric, (SAM) [25]. The SAM estimates the number of intersection tests performed by a ray traversing the structure using the following formula:

$$Cost(s) = Cost_{left}(s)P_{left}(s) + Cost_{right}(s)P_{right}(s)$$

Essentially, the expected cost of a particular partition is estimated to be the sum of the costs of the cells formed by the partition s. The expected cost of each cell is equal to the probability that a ray strikes that cell multiplied by the work performed by a ray striking that cell. These costs and probabilities are computed using some arguably inaccurate assumptions (studied in depth by Havran [29]). The heuristic based build process using this metric is known as the surface area heuristic (SAH) [25]. The SAH is a greedy, top-down algorithm that recursively chooses the best scene partition according to the SAM. In a later chapter of this dissertation, I present several changes to the SAH build process in order to improve build performance. I also present several novel cost metrics for hierarchical structures. The SAM isn't particularly effective when used to predict the amount of work performed in a uniform acceleration structure so additional metrics have been proposed for uniform acceleration structures by Ize [40].

2.2 Coherent Ray Tracing Background

Reductions in the cost of acceleration structure traversal combined with advances in CPU technology have recently [58] allowed for interactive rendering of static scenes using ray tracing on a single desktop machine (interactive performance had been demonstrated previously on a cluster [50]). Starting with Wald [72] and continued with Reshetov [57] and others [8, 70] an optimization known as packet tracing was the primary optimization developed to improve ray-tracing performance on modern CPU architectures.

Packet tracing [72] improves performance by traversing a group of rays through an acceleration structure at the same time. Packet tracing improves cache performance, the ratio of mathematical operations to branch operations and allows for efficient use of SIMD instructions. These features utilize key performance features of modern microprocessors to good effect. The fundamental feature exploited by this optimization is known as coherence. Geometrically, coherent rays have similar spatial extent and tend to intersect the same spatial partitions or geometric aggregates. Algorithmically, coherence is a combination of spatial locality in memory and instruction repetition (enabling the use of SIMD). Ray packets are primarily an optimization that improves machine utilization without reducing the amount of actual computation.

Packet-based ray tracing has also been improved by using an additional aggregation-based acceleration structure. Packets of rays may be represented by a conservative mathematical object such as a ray interval [11] or a bounding frustum [58]. By using a conservative representation of the packet, the entire

packet may be tested for intersection against an object or an object aggregate in one step. If the result of this intersection is negative then it is safe to assume that no rays within the packet could have intersected the object or object aggregation. Using conservative mathematical representations for ray-packets can reduce traversal and intersection overheads (in many cases) by replacing many individual ray tests with a single ray-packet test.

2.3 Specialized Structures Background

In addition to the general purpose acceleration structures discussed up until this point, many rendering systems use specialized acceleration structures. Specialized acceleration structures are acceleration structures that are specifically constructed with certain assumptions in mind. These assumptions can range from known scene or sample alignment to known temporal coherence properties of scene objects to known scene structure. Specialized acceleration structures are employed because they are often cheaper to build and traverse than their more general cousins and because they can enable additional optimizations.

The most common kinds of specialized acceleration structures are object-local acceleration structures. A rendering system may use many different object-local structures, each specialized for a particular object in the scene. These object-local structures use coordinate frames unique to the object they accelerate. By using local coordinate frames these acceleration structures implicitly consider the alignment of the objects they accelerate. This alignment

allows axis-aligned planes to more closely match the object's natural alignment which tends to significantly reduce the number of intersection tests performed by a ray passing near the object. In addition, object-local acceleration structures may be instanced. When using instancing, objects that occur multiple times in a scene are only stored once (along with their local acceleration structure) in memory. Instanced objects are stored as a reference along with a transform from the global coordinate frame into the specific object-local frame. When used effectively, instancing can reduce the amount of memory required to render a scene by orders of magnitude. The plants scene on the cover of PBRT [54] uses instancing to great effect.

Additionally, specialized acceleration structures can be used to align collections of rays (rather than objects) with a particular coordinate frame. The z-buffer [13], zz-buffer [62] and perspective grid algorithms introduced in this dissertation use the perspective transform in order to align rays to a coordinate axis. Under this transform, diverging rays with a common origin become parallel, axis-aligned rays. This alignment greatly simplifies ray traversal and is responsible for much of the performance in the algorithms that use the perspective transform. I will discuss the perspective transform in significant detail in a later section.

2.4 Fast-Build Background

In addition to adding traversal overhead to rendering, acceleration structures add overhead for their construction. Reducing the overhead as-

sociated with the construction of acceleration structures became a focus of much research in the past few years, driven by the requirement of scenes (and thus acceleration structures) to change between frames. Simple, fast-to-build acceleration structures such as the grid [70] and median split BVHs [48] presented an initial solution to the problem of dynamic scenes. However, these structures lacked the traversal efficiency of many of the more advanced, metric-based structures. The need to trace large numbers of rays for high-fidelity scenes spurred research related to making metric-based acceleration structure builds fast. Work published concurrently by the author [38] and by others [55] provided fast approximations for the SAM in order to improve the build performance of the SAH algorithm for kd-trees. The work presented in my paper is discussed in a later section of this dissertation. These ideas have subsequently been extended to BVH builders by Wald [67]. Approximations for the SAM reduce the overall computation required to build SAM based acceleration structures, thus improving performance.

Another common approach for improving acceleration structure build performance is to build acceleration structures lazily [19, 48] or to use a simpler acceleration structure (such as a uniform BVH) for small groups of objects [19, 57]. In this context, lazy-build algorithms are another term for demand-driven build algorithms. In contrast, non-lazy algorithms eagerly do work that may not be used in the result of a computation. This includes building an acceleration structure for a part of a scene that no samples pass through (perhaps another room not visible to the user). Lazy acceleration structure

build algorithms only build parts of the acceleration structure that are required to complete a rendering. Demand is provided by samples (rays) entering a partition or aggregation that has not yet been refined. Refinement (build) occurs only when demand arises. Many systems, such as those presented in papers by Djeu [19] and Reshetov [57] lazily build specialized acceleration structures.

Yet another approach for reducing build times is to use a pre-existing acceleration structure to build another one. Doing so can be as simple as updating all of the bounds in a BVH (to account for moving geometry) [48, 68, 79] or can involve a more detailed transcription process as described in my build from hierarchy publication [37]. I will discuss using a pre-existing acceleration structure to build a new (similar) acceleration structure in linear time in a later section of this dissertation.

2.5 Z-Buffer Visibility

Classical z-buffer rendering [13] (rasterization) is the most widely used 3D rendering technique. This technique is implemented in hardware by nVidia and ATI (among others) sold for use by high performance graphics applications and games. Z-buffer rendering is a highly-optimized special case solution of the general point-to-point visibility problem. Specifically, it is optimized for large numbers of regularly spaced queries with common origin, as is often the case for primary visibility. As I will discuss later, the performance of the z-buffer algorithm is almost entirely derived by its use of the perspective transform. Tiled rasterization [21] is a variant of the classical z-buffer algorithm where

image tiles are rendered independently. This independence requires geometry to be sorted and stored in separate bins.

As opposed to common ray tracing implementations, the z-buffer algorithm by itself is inefficient for high depth complexity scenes. The depth complexity of a sample is the number of scene objects it would intersect if it did not stop at the first. Specifically, the z-buffer does not use a sufficiently advanced sorting process to avoid testing all objects against a ray even when it only cares about the first. Thus the z-buffer algorithm behaves very much like a database "join." Because of this, most video games use an additional acceleration structure to cull the majority of scene objects that will not cause first intersections for any samples [66] (via view frustum culling, portals or other methods).

2.6 Other Perspective Transformed Visibility

In addition to the classic acceleration structures and the z-buffer, several other acceleration sturctures have been proposed over the years. One is the zz-buffer [62]. The zz-buffer uses a 2D uniform grid under the perspective transform (something I will discuss in detail later) to reduce the visibility problem to two dimensions. In two dimensions, the zz-buffer stores all geometry that overlaps each tile. Only rays and objects that land in the same tile are tested for intersection. The algorithm handles off-axis rays by conservatively enlarging bounding volumes of all objects such that off-axis rays must only check objects in the tile in which they start. This approach doesn't scale

well with largely off-angle rays as the conservative bounds end up covering a significant number of screen-space tiles. The zz-buffer has also unfortunately been all but forgotten in the ray tracing community and has not been used in any real-time ray tracing implementations.

Although it is not apparent from any publication, the VFX voxel tool [45] is rumored to use perspective space grids for volume rendering applications in movies.

2.7 Overview

This chapter focused on providing context for the data structures and algorithms contributed in my dissertation. As discussed, my research focuses on the visibility problem. In this chapter, I provided an introduction into the common techniques for solving the visibility problem, techniques which I build upon and extend. In particular I introduced acceleration structures for ray tracing and provided some intuition as to how they are used to reduce work required to compute visibility. I also introduced cost metrics and discussed their use in producing high quality acceleration structures. Several of the contributions in this dissertation relate specifically to cost metrics. Lastly I introduced perspective based acceleration structures and their advantages for common-origin ray tracing. I use the perspective transform in several of my ray-specialized data structures. To summarize, this chapter introduced concepts that I build upon throughout this dissertation.

Chapter 3

Visibility

This chapter begins the contributions of my dissertation. The remainder of the document will focus on the different the data structures and algorithms I developed over the course of my graduate studies and their contributions to the field of real-time visibility. To begin, I will describe the visibility problem at a high level. I will work from generality toward specifics, introducing my contributions as they become appropriate.

As mentioned before, visibility queries come in two forms:

1. "Given a scene and a ray, what scene object does the ray intersect first?"
2. "Given a scene and two points, is any scene object directly between the two points?"

From here forward, I will assume that rays terminate when they intersect an object. Rays intersecting semi-transparent objects spawn secondary rays in the same manner as they would when intersecting a refractive object. Using this assumption, rays never continue through objects, thus removing the need for a query considering objects other than the first.

As mentioned previously, the second query is less general and can be phrased in terms of the first. However, there are several important distinctions between the two that lead to very different characteristics. Specifically, the second query is unordered, has finite spatial extent and only requests a Boolean valued result. These properties allow for a significant number of optimizations for visibility queries of the second kind. Opaque shadow computations use this type of query.

3.1 A Taxonomy for Visibility Algorithms

Many different data structures and algorithms exist to accelerate the visibility problem. Understanding which structure and algorithm is best under which situation is a non-trivial exercise. In order to better understand different acceleration structures, I present a taxonomy based on common optimizations used by acceleration structures. This section outlines a collection of optimizations. The collection will form a basis with which I classify common acceleration structures and algorithms used in existing rendering systems as well as new structures I present in this dissertation. I intend for this classification to provide clarity of understanding regarding the character of acceleration structures. The bases of the taxonomy are commonly employed optimizations for visibility algorithms.

I will initially list and discuss the dimensions (optimizations) along which I will classify visibility acceleration structures and algorithms. It should be noted that any given rendering system can use multiple acceleration struc-

tures and traversal algorithms (even within one hierarchy). Therefore, this taxonomy cannot necessarily classify an entire rendering system but rather specific components of it.

Here I list the set of optimizations I have chosen as a basis. I chose these particular optimizations to be a balance between completeness and manageability. This set is by no means a complete list of possible optimizations.

- Organizational Strategy (partitioning/aggregation) - Acceleration structures use either partitioning or aggregation to accelerate the join. These were discussed in detail earlier.

- Change of Basis (many) - Many acceleration structures use a change of basis to improve performance. These changes of basis include object space transformations and perspective projection. I consider a locally transformed acceleration structure to be distinct from a global acceleration structure that happens to point to it. Using this distinction, each acceleration structure uses one frame of reference.

- Depth (hierarchical/uniform) - Hierarchical acceleration structures can provide a greater amount of adaptivity than uniform structures can. However, hierarchical structures are often substantially more complicated to build and traverse. Much information about a uniform acceleration structure is implicit and does not need to be explicitly stored or computed, resulting on simpler structures.

- Adaptivity (adaptive/non-adaptive) Data-driven data structures are often substantially more flexible than static algorithms. Taking cost metrics (such as the SAM) into account when building or traversing an acceleration structure can improve performance by providing the ability to tune the acceleration structure to specific rays and scene objects at run time. For example, a SAM based kd-tree is often significantly more effective than a median based kd-tree or oct-tree.

- Laziness (none/fine/coarse) - Demand driven (lazy) evaluation can improve performance in many cases. Laziness can occur at many different granularities. Fine-grained laziness does the minimal amount of work necessary to perform a task but can incur noticeable overhead for keeping track of what work has been done. Coarse-grained laziness eagerly computes parts of a result in order to reduce overhead.

- Streaming (samples/objects/neither) - Streaming is an optimization which elides storing either samples or scene objects (or both) in an acceleration structure. Instead, one collection is sorted and stored first, and then the other collection is sorted (often implicitly) and tested for intersection but never stored. Ray tracing typically streams rays and rasterization typically streams scene objects.

- Temporal Coherence (static/rebuild/refit) - Sometimes scene objects and samples change over time. Not all acceleration structures support such changes. Acceleration structures can support dynamic scenes and sam-

ples by rebuilding new acceleration structures each frame to account for this change. Alternately, acceleration structures can maintain a fixed topology aggregation hierarchy over time but refit the bounds of the aggregations to match changing objects.

As mentioned, this list is not intended to be a comprehensive list of all optimizations used in modern acceleration structures but rather to provide a basis within which we may classify common acceleration structures. In Appendix 1 I classify several common acceleration structures according to these optimizations. The ability to discuss the z-buffer algorithm cleanly in the context of a ray tracing taxonomy speaks to the power of this basis.

Another common optimization for visibility that I haven't listed in my taxonomy is approximation. Many visibility algorithms improve performance by solving a simplified problem. A classic example is shadow mapping, which reinterprets scene geometry as a collection of quads perpendicular to the view of a light. Another is geometric level of detail (simplification). I did not include approximation in the taxonomy because it isn't a true optimization in the sense of improving performance without changing the result of a computation.

By classifying data structures (and implicitly or explicitly their traversal algorithms) according to this taxonomy, we may correlate higher performance with specific optimizations and discover which provide the largest improvement in performance. Additionally we may derive new algorithms by combining these optimizations in ways that have not been previously explored.

This taxonomy provides significant context for the novel algorithms and structures I discuss later. It also highlights the source of the z-buffer algorithm's high performance and brings to light acceleration structures that allow for similar performance in the context of ray tracing.

3.1.1 Evaluation of the Basis

Having defined the taxonomy, I will now discuss the relative impact of each optimization. This discussion is intended to help provide insight into how the various optimizations effect rendering algorithms.

- Organizational Strategy (partitioning/aggregation) The organization strategy is arguably the least useful axis of this taxonomy. Despite at times almost religious fervor amongst the community about which strategy is superior, circa 2008 the community has acknowledged that neither is vastly superior. A good implementation of either will outperform an average implementation of the other.

- Change of Basis (many) - This optimization, on the other hand, is probably the most extreme axis of the basis. Changing the basis in which visibility is computed can have drastic implications in terms of performance (order of magnitude). The z-buffer gets its performance from a change of basis, as do all of my perspective space structures. Additionally, object space transforms can be used to avoid large amounts of memory usage (instancing) and avoid per-frame rebuilds for static objects undergoing ridged body transformations.

- Depth (hierarchical/uniform) - Uniform acceleration structures can have significant benefits in terms of both build and traversal performance. The z-buffer and perspective grid algorithms use this optimization to avoid traversal all together, greatly improving rendering performance. However, uniform structures in the general case have adaptivity problems and in general make poorer acceleration structures (the degree to which lack of adaptivity hurts is strongly scene-dependent and can range from negligible to unbounded).

- Adaptivity (adaptive/non-adaptive) Heuristic based algorithms get a lot of praise in the ray tracing community. Very few people argue in favor of simple (e.g. spatial median) split algorithms anymore. This stance is primarily due to the fact that algorithms like those presented in this dissertation have recently made heuristic-based algorithms "fast enough" that people have stopped bothering with inferior structures. It should be mentioned however that these data-driven algorithms are not always significantly better than simple ones. Over dense meshes for instance, median split does almost as well as the surface area heuristic. The perspective surface area metric [35] presented later in this dissertation demonstrates a significant improvement using adaptivity.

- Laziness (none/fine/coarse) Laziness is another optimization with potentially huge performance implications. My build algorithms in the Razor [19, 37] system demonstrate an order of magnitude improvement

in build performance due to laziness in many cases. It should be noted that laziness is a build optimization, not a rendering optimization. However, if acceleration structure build is less expensive, a system can spend time building more specialized or higher-quality structures, impacting rendering performance.

- Streaming (samples/objects/neither) - Streaming has no impact on the amount of computation performed when computing a visibility query. This optimization, however, does have an impact on the memory footprint of an algorithm which can in turn affect the cache performance of the specific hardware used. I discuss streaming in more detail in the cost metrics section of this dissertation. Streaming may become more important when costs of shading are added to the costs of visibility. It is important to note that streaming is **not** where the z-buffer algorithm gets much of its performance.

- Temporal Coherence (static/rebuild/refit) - Temporal coherence is similar to laziness in that it can potentially have a huge impact on build performance. As mentioned earlier, this improvement in build performance can be translated into rendering performance via better acceleration structures. The Build from Hierarchy [37] section of this dissertation uses temporal coherence (in the form of a scene graph) to improve build performance. (The scene graph in that system uses a refitting approach.)

This evaluation of the optimizations underscores the research directions I have taken over the course of my graduate studies as well as my working hypothesis.

When designing a visibility engine, the most effective (and potentially most complicated) optimization to take is the *change of basis*. It has the largest ramifications on rendering performance but can add significant build overhead. In general, when I refer to "specialized acceleration structures" I'm referring to a change of basis. In the Specialized chapter of this dissertation I demonstrate that change of basis is a cost-effective optimization even without the use of laziness or temporal coherence. When using lots of specialized acceleration structures, laziness and temporal coherence become high-priority optimizations. I also demonstrate these two optimizations to be extremely effective in the fast build section of this dissertation.

Using uniform acceleration structures can be a significant boon if rays or geometry are uniformly dense through the scene (as is the case for eye rays). However, hierarchical structures are important when scene or ray densities vary greatly. It is my belief that both uniform and hierarchical acceleration structures will be used at different resolutions within the same scene in order to obtain the highest performance. In this way, uniform structures are a specialization for more uniform geometry.

The adaptivity and organizational strategy optimizations are less emphasized in this evaluation. The effects of these optimizations are often significantly smaller than the others. Although they may have impact on the

performance, they are aspects of a system which are not usually difficult to change and decisions about them can often be late bound late during a system design.

Streaming as an optimization has either a huge or insignificant impact on performance, depending on the hardware being used. Streaming doesn't change the amount of mathematical computation but can have significant impact on the rendering performance of systems with constrained caches and/or memory bandwidth. I will discuss the impact of streaming in some more detail during my discussion of cost metrics. This process only gets more complicated when surface shading, multi-resolution, subdivision and displacement mapping are introduced. Solving memory-related problems for rendering is an extremely complicated issue and is, as of yet, poorly understood. To make matters worse, the hardware used for rendering is evolving rapidly at the moment (fall 2008) making this goal a moving target. In fact, algorithmic innovations with respect to streaming may even affect hardware design. Memory scheduling for rendering is already the topic of doctorial dissertations, and I believe it will continue to be in the future.

3.2 Overview

This chapter has discussed the visibility problem in more detail than the introduction did. In particular, I introduced a taxonomy for classifying visibility algorithms. This taxonomy is powerful enough to classify both ray tracing and z-buffering. Classifications are provided in the second appendix.

The taxonomy provides context by which to analyze rendering acceleration structures according to the optimizations they make. In addition to presenting the taxonomy, I have discussed the optimizations in the taxonomy in order to provide intuition as to the relative importance of each when designing new acceleration structures. Insight provided by this analysis motivates the perspective space acceleration structures presented in the specialization section of this dissertation.

Chapter 4

Cost Metrics for Acceleration Structures

Having introduced the visibility problem and discussed approaches for solving it and optimizations to those solutions, I will now proceed with a discussion covering cost metrics for acceleration structures. For those less familiar with acceleration structures, understanding cost metrics can illuminate many of the properties of the aggregation and partitioning approaches to accelerating visibility queries. As introduced previously, cost metrics are formulas that attempt to estimate the expected amount of computation required to compute visibility using a specific data structure and algorithm. Cost metrics have been well studied in the field of ray tracing [23, 25, 29].

4.1 The SAM

The most common cost metric for acceleration structures is the surface area metric (SAM). The surface area metric estimates the number of ray/object intersections an "average" ray will make when traversing the acceleration structure. The result of this approximation is equivalent to measuring the cost of the join in terms of the per-sample cost. At the macroscopic level, it is only reasonable to measure per-sample costs, because the number of sam-

ples is unknown until run-time. (However, other assumptions can be made for known sets of rays, such as eye-rays.) The surface area, as I will present it, assumes a binary acceleration hierarchy. Each node in the hierarchy represents a volume (axis-aligned box) of space. Internal hierarchy nodes have two children, each representing a different volume. In a kd-tree a split plane defines the two child volumes implicitly. In a BVH the child volumes are explicit. I will refer to the two child volumes as 'left' and 'right'.

Here is the formula for the SAM:

$$cost(s) = P_L(s)C_L(s) + P_R(s)C_R(s) \tag{4.1}$$

- P_L and P_R are the probabilities of a ray striking the left and right children respectively. These probabilities are conditional on a ray striking the volume defined at the node being evaluated. They are computed using ratios of surface areas between the children and the parent.

- C_L and C_R are the (usually approximated) costs of a ray intersecting the left and right partitions respectively. When evaluating an already constructed tree, these costs are obtained via a recursive invocation of the cost function. During a top-down (SAH) build process they are estimated to be the number of primitives overlapping each side.

The surface area metric works in the following way: the cost (expected number of ray-object intersection tests) for a ray striking a node in the hierarchy is equal to the probability of striking each child multiplied by the cost

of striking those children. Acceleration structure leaves have cost (expected number of ray-object intersection tests) equal to the number of objects they contain. The probability of a ray striking a child is estimated to be the ratio of the surface area of that child to the surface area of its parent.

The SAM makes several unrealistic assumptions. In order to compute intersection probabilities via ratios of surface areas, rays are assumed to have a uniform stochastic distribution of directions. Additionally, the costs of each child are assumed to be independent (this implies rays are assumed not to stop when they intersect an object). Some work has been done to address these deficiencies [29] but more accurate assumptions are either expensive to compute or reduce the flexibility of the acceleration structure and are therefore not often used. My work addresses some of these assumptions.

4.2 Details of the Visibility Join

Having outlined the most common cost metric, I will now present several details of visibility joins that have an effect on cost metrics. I will begin by discussing asymmetries between the two halves of the join. These asymmetries will have implications on the design of data structures and on the cost metrics for those structures.

A cross product between any two sets is an inherently unordered operation. A visibility join could treat samples and scene objects symmetrically. However, each group has several unique features that introduce asymmetries in practical solutions to the visibility join. I assert that the properties that

cause these asymmetries have been understudied to date and more careful understanding can lead to improved acceleration structure performance. Here I present a discussion of these properties and their effects on cost metrics for acceleration structures.

Samples have the following properties that scene objects do not:

1. Samples often have infinite (or large finite) spatial extent in one direction and small or no spatial extent in perpendicular directions.

2. Samples typically follow the flow of information in a rendering system. (Rendering results are more commonly associated with samples than with geometry and ultimately an image is produced from samples. Reyes style shading [17] is different in that it also deposits information on the geometry.)

3. Samples are often ordered from one end to the other. For example, eye rays are traced away from the eye. This property affects access patterns in the geometry among other things.

4. Samples are largely unknown at the beginning of the rendering process. This property forces pre-computed acceleration structures to be object-centric. Scene geometry can also be unknown during pre-processing and generated on the fly with subdivision and displacement. However, dynamic geometry tends to remain spatially localized, unlike secondary samples.

5. Some collections (eye, hard shadow) of samples are extremely well organized, originating at a single point or from a small volume. This organization is often far more uniform than any scene geometry. The z-buffer algorithm takes advantage of this uniformity.

6. Samples are extremely abundant. Commonly, $10^8 - 10^{10}$ unique samples (rays) per second are required to produce high-fidelity images at real time rates. This property induces a large focus on per-sample performance in rendering.

Scene objects have the following properties that samples do not:

1. Scene objects are often very spatially localized. Localization often allows scene objects to be easily bound into aggregations with reasonable spatial extent. (Subdivided geometry often exhibits even higher locality and a very high degree of uniformity.)

2. Scene information is highly compressed. Often large amounts of scene information are decoupled from visibility. Texture mapping, for example, removes much of the high-frequency information from a scene leaving a relatively low-frequency representation. Also, the level of information in a scene may easily be varied using subdivision and other level-of-detail schemes. Scene objects may also be simplified by bounding boxes or other proxies.

3. Scene objects are often organized for other purposes. Game engines typically track objects in a scene graph for purposes of view-frustum culling, animation updates and collision detection, among other things. This pre-organization can be helpful when building acceleration structures.

4. Scene objects have a very high level of temporal coherence. A scene typically changes very little from frame to frame and much of the geometry may remain completely unchanged over an extended period of time. This property allows geometry-centric acceleration structures to be re-used between frames.

The sample and scene object properties I listed all have implications on acceleration structures and rendering algorithm design. I will reference these properties in my following discussion of cost metrics for acceleration structures. They have an impact on the streaming optimization from the taxonomy I presented earlier and are the primarily influence on storage and flow of information within a system. They are therefore important when building a system that takes advantage of specific memory architecture. The implications of these features are far reaching. As I mentioned previously, entire dissertations have focused on memory issues relating to the visibility problem.

Classical ray tracing was designed with many of these properties (most likely implicitly) in mind. Recursive ray tracing (with a hierarchical acceleration structure) is designed to efficiently handle large numbers of lazily created samples. It is also capable of avoiding a large (circa 1985) memory footprint

by avoiding a frame buffer because it doesn't store per-sample information. Ray tracing also takes advantage of many of the properties of scene objects including spatial locality, representation of objects using simplified structures (bounding boxes) and perfect temporal coherence in non-dynamic ray tracing systems.

Classical z-buffering was also designed with some of these properties (implicitly) in mind. Specifically, the z-buffer algorithm hinges on the perfect regularity of the primary samples. Rendering systems using z-buffering also typically use large amounts of geometric compression based primarily on texture-mapping. Additionally, z-buffer engines typically use geometric aggregation (bounding boxes) to cull entire objects that do not intersect the view frustum.

4.3 Cost Metrics for Memory Usage

The visibility join has no inherent order. Samples and objects can be aggregated or sorted in any order. Ray tracing systems tend to aggregate or sort scene objects into an acceleration structure and traverse rays across that structure. Z-buffering, on the other hand, implicitly sorts all of the samples first and streams the scene objects across the samples. Here I will present several cost metrics for determining expected memory usage for various acceleration structures. These metrics can help researchers design data structures to achieve high cache utilization in cache-constrained systems.

Although the join is unordered in nature, ordering matters with respect

to memory footprint and access patterns. This dependency is caused by an optimization for the join that is so common that it usually isn't recognized as an optimization. When sorting two sets in order to perform a join, only one set needs to be stored in sorted form. (Both sets are always sorted even though this may not be obvious. E.g. ray traversal is a sorting procedure that doesn't store the results.)

First I will give an example of an algorithm that stores both samples and geometry. Given a grid structure, a scene and a set of rays:

1. Sort all of the scene geometry into the grid cells.
2. Sort all of the samples into the grid cells.
3. Visit each grid-cell and compute intersections for all samples and geometry in each cell (storing the nearest intersection along each ray).

This approach is, however, likely to be inefficient for a number of reasons. For this discussion I will focus on problems related to memory footprint.

An example of an algorithm that stores geometry and streams samples is traditional ray tracing. Geometry is sorted into and stored in an acceleration structure. Rays are streamed across this acceleration structure (traversal is a sorting process typically terminated upon first intersection) but not stored at the leaves. Instead the rays perform intersections and keep the results locally (in registers or on the stack).

An example of an algorithm that stores samples and streams geometry is z-buffering. Z-buffering implicitly sorts and stores samples in a grid. Objects are also implicitly sorted (rasterized) into the grid but thrown away after intersection tests are performed. Intersection results are stored in a per-ray data structure (the z-buffer).

Given a collection of samples and scene objects and a collection of spatial partitions (e.g. grid or BSP tree), it is possible to make an informed decision about which collection to sort and store first: scene objects or samples. The choice of collection doesn't affect the expected amount of intersection work performed to compute the join but can have a profound effect on memory performance. If both collections fit into cache, the order doesn't matter very much. However, if only one fits into cache, storing that collection and streaming the other avoids cache misses. If neither fits in cache, one of several things must be true to achieve optimal memory performance:

- Samples and scene objects both get coarsely sorted into bins and stored. These bins are recursively processed until one collection fits in cache.
- The collection of objects or samples being streamed is already coherent (i.e. partially sorted) in a way that makes hardware caching of the sorted structures effective. Ray tracing often has this property: rays that originate near one another tend to intersect the same objects, making effective use of hardware caching for both acceleration structure nodes and geometry.

- The join is sparse. When the join is sparse, each object or sample is only used a small number of times. This property implies that caching of any sort isn't going to be effective and "optimal" memory performance is poor. The z-buffer operates in this domain by assuming low depth complexity (each sample is only compared to a small amount of geometry).

4.3.1 Additional Considerations

Image information is associated with samples (not with geometry). If samples are sorted first, enough space must be allocated to store the results of the intersections (depth, color, etc.) along with or in place of the stored samples. In a z-buffer system, this storage is the depth (z) buffer and the frame buffer. In the same manner, when samples are streamed, the results must be streamed as output in order not to pollute the cache (the cache is assumed to be in use by the geometry). This property affects memory bandwidth requirements of the system.

Some system design characteristics can place restrictions on what can be streamed. In a recursive ray tracer with only one acceleration structure we have no choice but to sort scene objects first because we don't know all of the samples initially. We may however choose to stream scene objects across subsets of samples, an approach taken by packet tracing. Only by gathering collections of rays is it reasonable to stream geometry. Z-buffering uses this approach by implicitly gathering all of the eye rays in advance.

Additionally, the use of uniform acceleration structures, scene objects or samples can reduce the storage requirements for a particular collection of samples or scene objects substantially. For instance, the z-buffer algorithm uses a regular grid of samples that are stored implicitly and recomputed on the fly. This greatly reduces the required storage for rays.

4.3.2 The Metrics

Cost metrics can provide an estimate for the cache footprint of and bandwidth required by a particular visibility algorithm. Unfortunately, memory usage cost metrics generally fall into two categories: trivial and unpredictable. Unpredictable cost metrics vary heavily on the inputs to the join. For example: the number of nodes in a heuristic terminated kd-tree is impossible to predict for most scenes. This number can be bounded, but the bound isn't useful because the worst case is very bad. I will present several trivial memory usage metrics and demonstrate the benefits of even such trivial metrics on rendering performance. Throughout this section m will represent the number of samples and n the number of scene objects.

The standard BVH requires $(2n-1)*32+Kn$ bytes of storage including $(2n-1)$ internal nodes at 32 bytes each and Kn storage for the geometry itself. If we assume we have enough local storage for the BVH and that it is already loaded into that storage, using a BVH requires $6m$ bytes of output bandwidth (assuming 3 channels of half-float color output). This output bandwidth measurement assumes that display memory exists outside of the

local storage.

Alternatively, the z-buffer uses $10m$ (1 float depth field and 3 half-float color fields) storage and $Kn + 6m$ bytes of memory bandwidth (Kn to stream the geometry and $6m$ to flush the frame buffer). These measurements also assume that the depth and color buffers fit into the local storage and have already been loaded. The perspective grid behaves like a z-buffer when used to store parallel rays. These metrics are simple but provide an elegant tradeoff between memory bandwidth and cache requirements.

By using these metrics, acceleration structures can be designed to fit specifically into available cache. For example the tile sizes for the perspective grid acceleration structure (discussed in a later chapter) were chosen using these metrics to fit into available cache. By using these metrics, specialized acceleration structures can be chosen to fit into cache, ensuring low latency and high bandwidth access.

4.4 Total Cost Metrics

In addition to cost metrics for memory utilization we may extend cost metrics to include the total cost associated with a particular node. The total cost for a node is the cost of building the node plus the combined cost of traversing (through the node) all rays that intersect the node. Accumulating the total cost across all nodes doesn't result in a different amount of work than the traditional per-ray prediction (SAM value) added to the build cost of the structure but does clarify the distribution of that work. Specifically,

nodes high in an acceleration structure have significantly higher cost than do lower nodes. Depending on the build algorithm, some of this work can come from the build process, but much of it comes from traversal. The upper levels of a hierarchy are touched significantly more often than lower levels. Most obviously, in the majority of traversal algorithms, the root node gets touched by every ray (or packet) that touches the acceleration structure.

This section does not contain explicit formulae for two reasons. First, it is difficult to predict the number of rays striking a specific node to even a moderate degree of accuracy. Second, the large variety in build costs due to different algorithms makes predicting build costs very difficult and ever changing. Hardware details such as cache sizes and behaviors further complicates cost prediction. Regardless, I believe the following discussion has pedagogical benefits.

The relative cost of traversal is extremely high for upper levels of the acceleration structure. Any amount of additional per-ray overhead associated with a particular acceleration structure is magnified at these points in the hierarchy. As an example, I will discuss the difference between a kd-tree and a BVH. Kd-trees tend to have more nodes than BVHs but do (many times) less work per node. The overhead of more nodes increases memory usage and adds additional unpredictable branches to the traversal process. Interestingly, these differences seem to mostly cancel each other out and the two acceleration structures perform similarly well in the average case. However, in the upper levels of a hierarchy, a kd-tree traversal algorithm will perform significantly

less math than a BVH will over the top portion (e.g. the top 10 levels) of the acceleration structure. BVHs tend to perform better at the lower levels of an acceleration structure, due largely to the fact they avoid the "integral duplication" problem described in the next section and fixed (lower) number of tree nodes. This analysis largely suggests that kd-trees are a more appropriate coarse acceleration structure than BVHs.

It should be noted that 10 levels of a BVH can potentially produce a more refined acceleration structure than 10 levels of a kd-tree because it uses 6 times as many planes. However, half of the planes in an traditional BVH are duplicated between each child-parent pair, significantly reducing the number of *unique* planes in a BVH. The result is that a BVH does 6 times the work of a kd-tree and can have at most 3 times as many unique planes. For large numbers of rays, the work/refinement ratio still favors the kd-tree at the top of a hierarchy.

4.5 Metrics Continued: Corrections to the Surface Area Metric

In addition to the prior discussion of cost metrics, I present a correction to the original surface area metric in order to account for a specific optimization (mailboxing) and a new metric with more accurate assumptions for specialized acceleration structures. The latter is rederived from the ground up using new assumptions about ray distribution for an acceleration structure.

4.6 Corrections to the Surface Area Metric with Respect to Mailboxing

The work presented in this section draws primarily from my work published at the IEEE Conference on Interactive Ray Tracing 2008 [34].

As mentioned previously, the surface area heuristic is the de-facto standard algorithm for producing high-quality adaptive acceleration structures. This algorithm is used to produce acceleration structures for both high-quality and real-time ray tracing applications. Hhigh-quality acceleration structures minimize the per-ray cost of ray tracing, reducing overall render times. With recent advances in acceleration structure build algorithms, these per-ray costs are continuing to dominate time spent ray tracing, even for real-time applications. Given the importance of per-ray costs in ray tracing, improvements to the surface area heuristic are widely beneficial.

In particular, it is well known that the surface area metric (SAM) makes several unrealistic assumptions. These assumptions include assuming uniform distribution of ray direction and assuming that rays skewer (pierce geometry without intersection) the scene. Havran's thesis [29] describes a more general cost metric which corrects for several of the inadequacies of the initial metric. However no previous work addresses the interaction between the surface area metric and another common ray tracing acceleration algorithm, mailboxing. This section addresses the interaction between mailboxing and the surface area metric. I provide a corrected cost metric for use in systems using mailboxing. Additionally, I provide an example of how this correction fixes a problem with

kd-trees. Finally, I show that this simple correction to the surface area metric can lead to noticeable performance improvements in ray tracing systems that use mailboxing.

4.6.1 The Mailboxing Optimization

Mailboxing [29] is a ray tracing optimization specific to spatial partitioning acceleration structures. Due to the fact that an object can potentially fall onto both sides of a spatial partition, a ray that intersects both sides of the partition can potentially attempt to intersect the same object multiple times. These additional intersection tests are entirely redundant. Mailboxing is a method of tracking intersections in order to remove these redundant computations.

Traditionally, mailboxing is implemented by adding a "mailbox" to each object in a scene. This mailbox is a small allocation of memory that stores the ID of the last ray that tested against the object. If a ray visits an object more than once, its ID is already present in that object's mailbox and no redundant intersection is performed. In a packet-based ray tracer, a packet ID is stored instead of a ray ID, and whole packets are checked at the same time. Therefore, the overhead of mailboxing is amortized across the packet. Wald [70] describes significant performance advantages for mailboxing using packets and the coherent grid traversal algorithm.

However, classical mailboxing suffers from poor cache performance on modern architectures and has at points been evaluated and considered more

harmful than helpful [30]. Even moderate-sized scenes can have several megabytes of mailboxes that can be accessed in a non-coherent order, causing huge problems for hardware caches.

To address these memory-related concerns, variations of classical mailboxing have been developed to mitigate the problem of having a large read/write data structure. Hashed mailboxing [8] keeps a small hash table of the last several intersections in order to reduce storage requirements. Hashed mailboxing has been shown to have performance benefits comparable to those of classical mailboxing while using only a small fraction of its memory. Recently, Shevtsov [51] introduced *inverse mailboxing*, which uses a significantly smaller amount of memory than even hashed mailboxing. Inverse mailboxing keeps a cache of recently visited object IDs for the current ray packet (as opposed to keeping recently visited ray IDs for each object). The cache is conservative and doesn't necessarily eliminate all redundant intersection tests but in practice eliminates many, even with a small 8-entry cache. The paper shows that inverse mailboxing approach provides performance improvements in a real-time ray tracer proving that mailboxing has potential to be a useful optimization in modern systems.

4.6.2 Mailboxing and The SAM

Since it conditionally removes some of the intersection tests performed during traversal, mailboxing has an effect on the number of expected intersection tests performed when traversing an acceleration structure (and thus

the cost of the structure). This reduction should be reflected in the metrics used to pick scene partitions. Specifically, rays that traverse both sides of a partition should not be considered to intersect duplicated geometry more than once. Taking this reduction into account changes the SAM to the following:

$$cost(split) \quad = \quad C_S + C_L P_L + C_R P_R - C_{L \wedge R} P_{L \wedge R} \tag{4.2}$$

- $C_{L \wedge R}$ is the amount of work that is duplicated on both sides of the partition. Using the common recursive cost estimate (number of objects on each side) this value is the number of objects that overlap both sides of the partition.

- $P_{L \wedge R}$ is the probability that a ray traverses both sides of the partition. A ray strikes both sides of the partition only if it strikes the partition dividing the two child cells. Since this partition is a convex object contained within the parent cell, the probability of hitting it may be computed as a ratio of surface areas just like P_L and P_R.

This simple modification to the SAM corrects for mailboxing in the context of SAH based kd-trees. It has an impact on the construction of kd-trees relative to BVHs. In the implementation chapter I describe its effect in a common geometric scenario and discuss why the new kd-tree has lower cost. In short, the analysis shows that the additional work induced by object duplication is addressed by the mailboxing optimization, but without a change to the cost metric, the potential benefits of mailboxing aren't fully realized.

4.7 The Perspective Surface Area Metric

As mentioned previously, the surface area metric makes several unrealistic assumptions. These assumptions are even less realistic in other spaces. Many of the ray-specialized acceleration structures I describe in this dissertation use perspective space. The surface area heuristic doesn't build very effective hierarchies in perspective space because of its assumptions about ray distribution. These poor assumptions are exacerbated under the perspective transform. In this section I will re-derive a new metric, which I will refer to as the Perspective Surface Area Metric (or PSAM for short). I will demonstrate that the PSAM provides higher-quality acceleration structures in perspective space. Additionally, if an acceleration structure is being constructed per frame per light (as is the case for my results using perspective structures), more accurate assumptions can be made.

The assumption of uniform ray direction is, in some cases, a reasonable assumption. In the past, acceleration structure build was an offline process. Acceleration structures were built once and then potentially reused for many different camera positions. This scenario provided a wide range of potential rays to be traced using a given acceleration structure. More accurate assumptions, however, provide higher-quality acceleration structures for given sets of rays. This problem was studied in detail in Havran's dissertation [29], and many modifications were provided to address these issues. However, at the time, per frame rebuild was considered to be too costly to be practical and many of the proposed improvements have not been used for general-purpose

ray tracing.

Recently, in order to support dynamic scenes, researchers have successfully made acceleration structure build an online process. Dynamic ray tracing systems rebuilt or refit an acceleration structure (or in some cases many structures) each frame [19, 48, 57, 68, 79]. This per frame rebuild allows for restrictions to be placed on the acceleration structure such as known camera or light locations. The PSAM will make use of these assumptions.

4.7.1 Brief Introduction to Perspective Space

I define perspective space as world space transformed by the perspective transform. The perspective transform is defined by the following equations:

$$x' = x/z \tag{4.3}$$

$$y' = y/z \tag{4.4}$$

$$z' = -1/z \tag{4.5}$$

The perspective transform is a non-affine transform that map lines to lines (precisely, it is a projective transform). Since this transformation maps lines to lines, it maps rays to rays and polygons to polygons. Additionally, ray tracing in perspective space is identical to ray tracing in world space. An acceleration structure built in perspective space can be traversed by rays in perspective space using all of the same algorithms that one would use in world space. Even ray-triangle intersection can be performed in perspective space, although barycentric coordinates must be corrected prior to shading.

4.7.2 Adaptive Perspective Space Acceleration Structures

The oft erroneous assumption of uniform incoming ray direction made by the SAM allows the probability of intersecting a child node to be computed as a ratio of node surface areas. These node surfaces are usually axis-aligned boxes, whose areas are inexpensive to compute. To use this metric in perspective space, the node faces must be transformed back into regular space before computing their surface areas. These transformed faces are in general not axis-aligned and their areas are thus more expensive to compute. Alternatively, if we changed the assumption to be: "incoming ray directions are uniformly distributed in *perspective* space", then we could use surface areas computed in perspective space from axis-aligned boxes. However, the assumption of uniform incoming ray direction in perspective space is likely to be even less accurate than the assumption of uniform incoming ray direction in world space. Either way, the assumption of uniform incoming ray distribution in any space is unreasonable when the acceleration structure is used for just one frame with either one light or one camera.

I opt for a first principles re-derivation of a heuristic for perspective space acceleration structures. First I abandon the assumption of a directionally uniform incoming ray distribution in favor of a distribution more appropriate for cameras and area lights. Second, I compute the probability of hitting a node in perspective space.

The new metric (the perspective surface area metric) more accurately models the distribution of rays that use a perspective space acceleration struc-

ture. The metric assumes that ray origins (or destinations) have a uniform distribution on the surface of an axis-aligned quad (which will be referred to as the **aperture**). The metric also assumes a uniform distribution of ray directions leaving one side of the aperture. More specifically I define a uniform directional distribution as meaning that if we place a plane at a certain distance from the origin, the spacing of ray intersections on the plane will be uniform. This distribution is equivalent to the equal spacing of pixel centers on an image plane. Figure 4.1 illustrates this distribution.

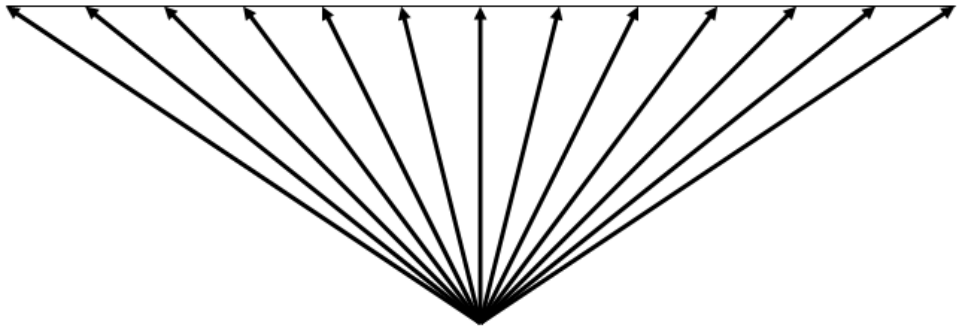

Figure 4.1: A cross section of rays with a "uniform" distribution in direction (by the assumptions). These rays are **not** uniformly distributed in angle, but this is the common distribution for eye rays. This style of uniformity also translates into perspective space nicely.

The assumption that ray origins are uniformly distributed is almost always accurate given common sampling patterns associated with Monte-Carlo integration. The assumption that outgoing ray directions are uniformly distributed is somewhat less accurate, but I use it anyway for two reasons. First, it is impossible (or at least very difficult) to know the distribution of secondary rays within a scene a priori. I do not propose solving that problem here. The

second and perhaps more comforting reason is that the probabilities computed by the PSAM using this assumption are ratios. Given that we are computing ratios, as long as the distribution of rays in the direction of the parent box is locally close to uniform, the effects of variation in ray density will cancel out and the computed ratios will be close to accurate.

Before I begin the derivation of the PSAM, I would like to outline some properties that it intuitively should have. First, if the area of the aperture is zero (i.e. the light is a point light) then the probability of a ray striking a box should be proportional to the projected area of the front face of the box. That is, the probability of striking the box should be proportional to the *area* of the box when projected into two-space. Second, moving twice as far away from a light should have the same effect as shrinking the light by one half in each dimension (following the rule that asymptotically, light falls off quadratically with increasing distance). In other words, area lights should appear smaller the farther away from them you get. In the limit, partitions that are far from an area light should be the same as partitions from a point light.

4.7.3 Derivation of the PSAH

For the new cost metric I use the same form as the traditional SAM:

$$cost_{traversal} = c_{node} + \sum_{children} P_{child} C_{child} \tag{4.6}$$

The cost estimates will remain the same, simply the number of objects overlapping each child. This section will be dedicated to deriving new prob-

ability terms using the assumptions described earlier. The probability of a ray striking a child box given that it struck the parent box is the ratio of the number of rays (out of all possible rays) that strike the child box divided by the number of rays that strike the parent. The number of rays that strike a box may be formulated as an integral over all rays of the Boolean intersection function for a ray/box pair. The probability of striking a child box is the ratio of these integrals.

$$p(child) = \frac{\int_{Rays} hit(child, ray) dray}{\int_{Rays} hit(parent, ray) dray} \tag{4.7}$$

In order to more accurately define these integrals, I formalize several of the assumptions about the distribution of rays that use a PSAM based acceleration structure. Rays are assumed to launch from a rectangular aperture A and are assumed to have a uniform distribution of slope. Rays will take the following form:

$$ray := (o, d) \tag{4.8}$$

$$o \in [-A_x, A_x] \times [-A_y, A_y] \times [0] \tag{4.9}$$

$$d \in (-\infty, \infty) \times (-\infty, \infty) \times [1] \tag{4.10}$$

Where o is the ray origin taken from a uniform distribution on a rectangle at $z = 0$ and d is a direction taken from a uniform distribution of intersections with a plane at $z = 1$. See figure 4.1 for a two-dimensional cross

section of this distribution in direction.

$$x = d_x t + o_x \tag{4.11}$$

$$y = d_x t + o_y \tag{4.12}$$

$$z = t \tag{4.13}$$

Then we transform these equations into perspective space:

$$x' = x/z = (d_x t + o_x)/t \tag{4.14}$$

$$y' = y/z = (d_y t + o_y)/t \tag{4.15}$$

$$z' = -1/z = -1/t \tag{4.16}$$

By defining a new parametric variable for the ray, $t' = 1/t$, we can rewrite the perspective equations as:

$$x' = o_x t' + d_x \tag{4.17}$$

$$y' = o_y t' + d_y \tag{4.18}$$

$$z' = -t' \tag{4.19}$$

Notice that the roles of values for o and d have been reversed in perspective space. By introducing new variables $o' = d$ and $d' = o$ representing the ray origin and direction in perspective space we get:

$$x' = d'_{x'} t' + o'_{x'} \tag{4.20}$$

$$y' = d'_{y'} t' + o'_{y'} \tag{4.21}$$

$$z' = -t' \tag{4.22}$$

And from the swap of o and d and their initial distributions we also obtain the following distribution for o' and d':

$$ray' := (o', d') \tag{4.23}$$

$$o' \in (-\infty, \infty) \times (-\infty, \infty) \times [0] \tag{4.24}$$

$$d' \in [-A_x, A_x] \times [-A_y, A_y] \times [-1] \tag{4.25}$$

A common confusion regarding the above transform is that many readers think about transforming an origin and a direction separately. Thinking about rays in this manner leads to questions regarding limits or infinity because the origin in "normal" space has $o_z = 0$. Rather, I would recommend thinking about rays as line-equations put through a perspective transform. We may now more accurately specify the intersection integral over all rays in perspective space:

$$\int_{-A_x}^{A_x} \int_{-A_y}^{A_y} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} hit(box', ray') do'_{y'} do'_{x'} dd'_{y'} dd'_{x'} \tag{4.26}$$

In perspective space, if we fix a ray direction (from $[-A_x, A_x] \times [-A_y, A_y] \times [-1]$) and assume a uniform distribution of origins, the number of rays that strike a box is equal to the area of that box projected onto the $z' = 0$ plane in the direction of the ray. See figure 4.2. This area may be computed by summing the projected areas of the three visible faces from the fixed direction. See Figure 4.3 for reference with regard to the following equations. Given a

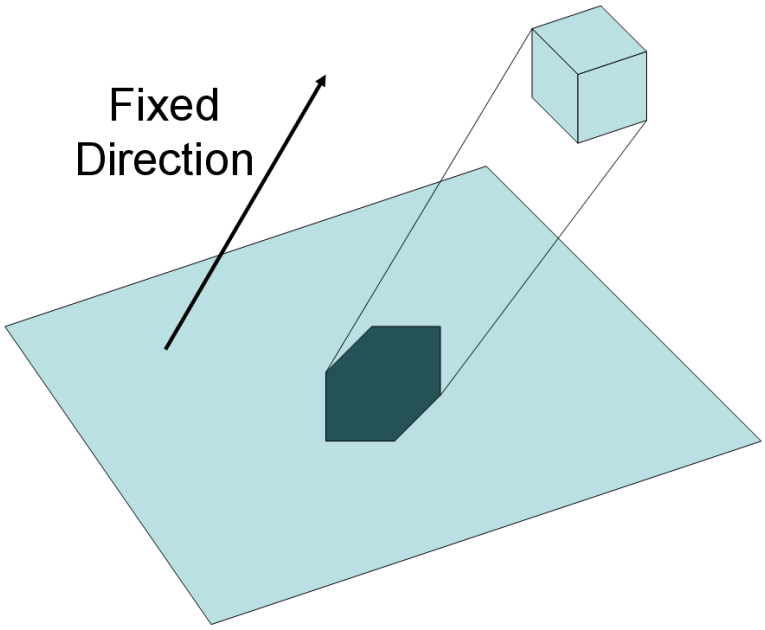

Figure 4.2: Given a fixed direction (indicated) the shadowed region is the set of origins that will intersect the box. The area of the shadowed region is computed in Equations (4.27)- (4.29).

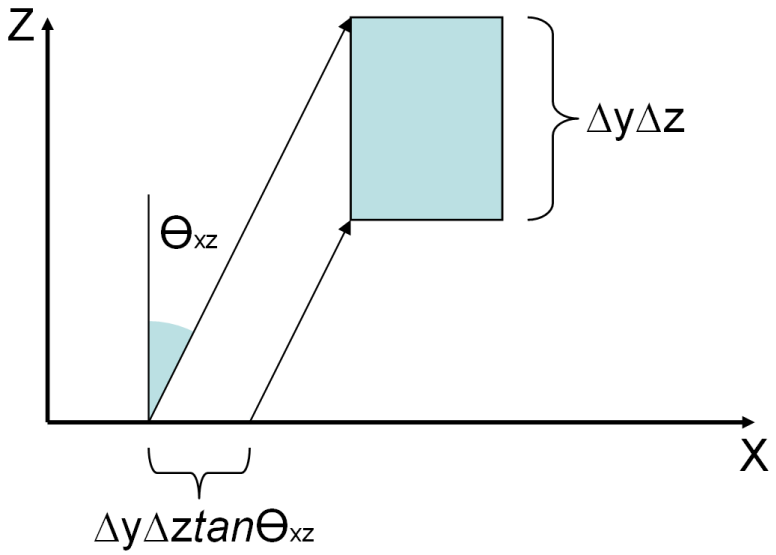

Figure 4.3: A cross section of a box being projected onto a plane. This figure is designed to give intuition into equation (4.28). The planes of the box perpendicular to the $z = 0$ plane have areas scaled by the tangent of the direction vector.

fixed ray direction we have:

$$area = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} hit(box', ray')do'_{y'}do'_{x'} \tag{4.27}$$

$$= \Delta x'\Delta y' + |\tan\theta_{y'z'}|\Delta x'\Delta z' + |\tan\theta_{x'z'}|\Delta y'\Delta z' \tag{4.28}$$

$$= \Delta x'\Delta y' + |d'_{y'}|\Delta x'\Delta z' + |d'_{x'}|\Delta y'\Delta z' \tag{4.29}$$

Where $\Delta x'$, $\Delta y'$ and $\Delta z'$ are the dimensions of an axis-aligned box in perspective space. Substituting this formula into the original integral we get:

$$\int_{-A_x}^{A_x}\int_{-A_y}^{A_y} \big(\Delta x'\Delta y' + |d'_{y'}|\Delta x'\Delta z' + |d'_{x'}|\Delta y'\Delta z'\big)dd'_{y'}dd'_{x'} \tag{4.30}$$

$$= 4\int_{0}^{A_x}\int_{0}^{A_y} \big(\Delta x'\Delta y' + d'_{y'}\Delta x'\Delta z' + d'_{x'}\Delta y'\Delta z'\big)dd'_{y'}dd'_{x'} \tag{4.31}$$

$$= 4\int_{0}^{A_x} \big(A_y\Delta x'\Delta y' + \frac{A_y^2}{2}\Delta x'\Delta z' + A_y d'_{x'}\Delta y'\Delta z'\big)dd'_{x'} \tag{4.32}$$

$$= 4\big(A_xA_y\Delta x'\Delta y' + \frac{A_xA_y^2}{2}\Delta x'\Delta z' + \frac{A_x^2A_y}{2}\Delta y'\Delta z'\big) \tag{4.33}$$

$$= A_xA_y(\Delta x'\Delta y' + \frac{A_y}{2}\Delta x'\Delta z' + \frac{A_x}{2}\Delta y'\Delta z') \tag{4.34}$$

Due to the fact that probability is a ratio of these integrals, the division will cause common factors to cancel. Thus we may scale the computed integral to:

$$G(box, A) \;=\; \Delta x'\Delta y' + \frac{A_y}{2}\Delta x'\Delta z' + \frac{A_x}{2}\Delta y'\Delta z' \tag{4.35}$$

This expression is similar to the traditional SAM except that it contains scaled aperture terms. With it we may compute the probability of striking each

child box and thus the cost of a partition:

$$cost_{traversal} = c_{node} + \sum_{children} \frac{G(child, A)}{G(parent, A)} C_{child} \tag{4.36}$$

With a solution in hand, let's look back at the intuitive properties we required in an appropriate cost. First, if the area of the aperture is zero (e.g. a light is a point light) than the probability of striking a box should be proportional to the projected area of the front rectangle of the box. Second, if we move twice as far away from a light, it should have the same effect as shrinking the light by one half in each dimension. Both properties are clearly provided by the solution formula. Setting $A = (0, 0)$ removes the $\Delta z'$ terms and yields $\Delta x' \Delta y'$. Increasing z_{near} and z_{far} by a factor of two decreases z'_{near} and z'_{far} by a factor of two and thus $\Delta z'$ by a factor of two. Since $\Delta z'$ always occurs multiplied with A_x or A_y, scaling z has the same effect as dividing A by two. Due to these effects, parts of the scene with large z values will not choose splits in z' nearly as often as those in x and y because those terms will dominate the cost function.

This cost function has the useful practical properties that it is simple to compute, is linear in x', y' and z' and requires very little algorithmic change from the original SAM to adopt. The perspective transform and the PSAM provide a significant improvement over the origins SAM and world space. As the implementation chapter will discuss, near common origin ray tracing is approximately 30% faster in using perspective space and the PSAM than it is in world space using the SAM. For large collections of rays, this improvement

in rendering performance usually outweighs the additional cost of building a specialized acceleration structure. The results in the implementation chapter also demonstrate that the PSAM is twice as effective as the original SAM in perspective space, justifying all of the effort required to derive the new cost metric for perspective space.

4.8 Concluding Remarks

I have spent this chapter discussing cost metrics for acceleration structures. Cost metrics (typically) estimate the cost of traversing an acceleration structure. They are used in build algorithms (e.g. surface area heuristic) to make locally optimal decisions by providing a basis of comparison for candidate scene partitions.

In this chapter, I first provided background and discussed properties of the visibility problem that are relevant to the design of cost metrics. After introducing the topic, I presented cost metrics for memory bandwidth and footprint and discussed metrics to measure combined build and traverse cost. I also discussed a correction to the surface area metric to account for the mailboxing optimization. Lastly, I also presented a new cost metric for acceleration structures under the perspective transform. This transform, along with the assumption that a post-transform structure only accelerates one light or camera, changes enough assumptions to warrant a new cost metric. In the implementation chapter I will show that the new metric is twice as effective as the SAM in perspective space.

Cost metrics are an important aspect of acceleration structure research. Acceleration structures created using cost metrics are often significantly faster to traverse. Additionally, cost metrics provide useful incites when designing acceleration structures by measuring potential workloads. The metrics and corrections I presented in this chapter contribute to the understanding and effectiveness of acceleration structures.

Chapter 5

Specialized Acceleration Structures

Given the arguments for specialized acceleration structures (change of basis and temporal coherence) presented in the taxonomy section, it should not be surprising that all rendering systems used in video games use multiple acceleration structures. A z-buffer is used for visibility and "shadow" queries. BSP trees or something similar are commonly used to store static scene geometry and object hierarchies are used to manage dynamic models and collision detection. To further my hypothesis about specialized acceleration structures, I assert that all of the following cases are best handled each with their own specialized acceleration structure:

- Static scene geometry/objects - Highly optimized acceleration structures can be created off-line for static geometry. Static geometry exists in all flavors from buildings to objects that don't change form, such as a sword or a table.

- Dynamic objects with static topology - Most non-static objects fall under this category. Skinned characters and other deformable objects are common examples. Acceleration structures (BVHs) with static topology can often effectively accelerate these objects without ever being rebuilt.

Although the trend in games is to make the world more and more dynamic, many objects are fundamentally static (landscape and buildings are two common examples).

- Dynamic collections - Characters and particles often have large dynamic ranges of movement within a scene and can require fully dynamic acceleration structures. However, the number of dynamic objects that need to be tracked in such a structure is often orders of magnitude smaller than the number of primitives making up those objects.

- Coherent Visibility Queries - Large numbers of coherent visibility queries are often best accelerated with their own acceleration structure. The z-buffer is a classic example of such an acceleration structure. The perspective space acceleration structures I present also fall into this category.

- Global Illumination - Acceleration structures for global illumination may look very different than they do for high-frequency visibility.

This list should help reiterate the point that there is no single best acceleration structure. Many different structures are needed for the diverse array of geometry and queries. Although the idea is not new [6], at this moment the idea of using multiple acceleration structures in the context of real-time ray tracing is not widely practiced. A common concern when using multiple specialized acceleration structures is the cost of building and maintaining such structures. One should note that many of these cases don't in fact require a

structure rebuild. Static geometry requires no update at all and structures (BVHs) with static topology only require a bounds update (an operation that is known to be fast). In this section I demonstrate several algorithms that show that fully dynamic builds can also be fast. In addition I demonstrate that using multiple (high-quality/cost) specialized acceleration structures for primary visibility and shadows can outperform general-purpose acceleration structures.

It should be noted that multiple acceleration structures can interact in different nontrivial ways. Games often use a BSP tree or portal system to cull geometry before feeding it to a z-buffer. Such a system used in conjunction with z-buffering is an example of two acceleration structures being used in tandem. Some acceleration structures are simply the leaves of other structures. For example, PBRT uses object space acceleration structures that occur at the leaves of a world-space structure. I will demonstrate an algorithm that uses one acceleration structure as input to improve build performance for another acceleration structure.

5.1 Geometry Specialized Acceleration Structures

One of the most common specializations for acceleration structures is geometric. Geometry-specialized acceleration structures are constructed with a specific set of geometry in mind. These structures can take advantage of specific features of the geometry in order to improve build or render performance. These features include object alignment and temporal coherence.

5.1.1 Object-Space Structures

Object-space acceleration structures are geometry-specialized structures. Many of them are constructed, each over a single object (or small group of objects). These structures use a coordinate basis in which the object is "aligned" [54]. Some objects don't have natural alignment, but the majority does. A quick look around a room will reveal a host of objects with a specific natural alignment (a book, a chair, even people). By constructing an acceleration structure in the basis of the object, axis-aligned bounding boxes and spatial partitions more closely match the contour of the object being accelerated. By having a specific structure for each object in the scene, visibility queries for each object are performed (more efficiently) in their own space. Object space acceleration structures are typically supported by inserting them in their entirety into a world-space acceleration structure as a leaf. In this way, acceleration structures are layered on top of one another, where the leaves of one structure are the roots of others.

Interestingly, the majority of acceleration structures have some object alignment because artists tend to produce scenes that are aligned in some way to coordinate axes. Most predominately, the ground in a scene is oriented perpendicular to a specific (usually y) axis. Although few realize it, this property actually greatly improves rendering performance for many scenes. A large difference in performance can be observed by simply rotating a scene and camera. The difference can be a large factor and is due solely to geometry-scene alignment. The improvement in performance comes from extremely tight bounds

on objects when aligned to the coordinate axes. Imagine a scene with a floor plane implemented as a single pair of triangles. This plane is aligned perpendicular to one coordinate axis of the scene and has extremely tight axis-aligned bounds. Rays must break a triangle's bounding box before intersecting that triangle. However, since these bounds are very tight, only rays that pass very close to the floor will tested for intersection against it. Alternatively, consider the same situation but with the scene alignment such that the long edges of the triangle are aligned to the vector $(1, 1, 1)$. In this case, the axis-aligned bounding volume for each triangle is extremely large. Rays that don't pass near the floor will still be tested for intersection against the floor plane because of these bad bounds stemming from scene misalignment. I encourage people to perform this experiment themselves to make it clear that object-specific alignment is important for high performance when axis-aligned planes are used.

5.1.2 Topology-Specialized Structures

Object-aligned acceleration structures are not the only kind of geometry-specialization in acceleration structures. Topology-specialized acceleration structures are specialized to a specific geometric topology. This topology is assumed not to change over the life of the acceleration structure. These structures are typically aggregation-based structures (BVHs), because refitting a BVH to a dynamic (constant topology) object is an extremely simple procedure. Partitioning-based acceleration structures can also work in this domain

as demonstrated by the fuzzy-kd-tree [27]. Topology-specific structures are an easy way to maintain a high-quality acceleration structure without the need to rebuild it each frame.

A special case of topology-specialized structures are implicit topology structures. Dynamically created geometry (most commonly subdivision surfaces) is often created with an implicit topology. This topology can be imposed onto an acceleration structure without any difficult work. Specifically, building high-quality BVHs over Catmull-Clark diced grids can take advantage of the implicit layout of the grid when building a topology-specific acceleration structure for the grid. The Razor project [19] (on which I worked) uses this approach for accelerating subdivision grids. In fact, these grids are simple enough to create that the Razor system sometimes discards them after use and rebuilds them if necessary without a noticeable impact on performance.

5.2 Ray-Specialized Acceleration Structures

In addition to specializing acceleration structures for geometry, they can be specialized for large groups of rays. A major contribution of this dissertation is the introduction and analysis of two ray-specialized acceleration structures, the perspective grid and the perspective kd-tree (using the perspective surface area metric). Ray specialization is in fact very similar to geometric specialization. The idea is to perform a transform such that rays align themselves along a coordinate axis. By aligning rays to a specific axis, ray traversal is restricted primarily to one dimension, allowing a number of

optimizations. It is important to note that no transform is likely to arrange all of the rays in a scene along a particular axis. Therefore, ray-specialized acceleration structures are often constructed on collections of rays with common properties. Examples of large numbers of rays with common properties are primary rays (eye and depth of field rays) or shadow rays (hard and soft). It is not uncommon to have over one million primary rays with the same origin.

Since each collection has a different transform, each requires its own acceleration structure. Therefore, a system using ray-specialized acceleration structures will use at least one structure for primary visibility and likely an additional structure for each light source in the scene. Despite the apparent cost of building and maintaining such a large number of acceleration structures, In the implementation chapter I demonstrate that the improvement in traversal performance more than makes up for the additional build cost. This result shouldn't be surprising to anyone familiar with shadows and visibility using z-buffer hardware. In such a system each light has a different shadow map and the camera also has its own depth buffer.

Not all ray-specialized acceleration structures specialize on the scale of millions of rays. Reshetov [57] introduces an algorithm for faster packet intersection via frustum culling. This frustum is an ephemeral ray-specialized acceleration structure built to quickly cull geometry against a packet before performing intersection tests.

5.2.1 Face Culling

A specific feature of ray-specialized acceleration structures is that rays using the structure have a known specific distribution of directions. When using manifold geometry and a ray-specialized acceleration structure, geometric faces can be sorted into front-facing and back-facing. Rays cannot intersect with back-facing faces without first intersecting a front facing face. Thus for primary visibility, all back-facing faces may be ignored without changing the result of the visibility query. Face culling can be effectively used to reduce the number of objects stored in an acceleration structure (by about half). This optimization is very commonly used in z-buffer renders.

In the case of shadow rays, front-facing and back-facing faces produce identical silhouettes (and thus the same shadows). This silhouette property is obvious when one realizes that the silhouette is defined by the boundary between the two sets. When tracing shadow rays, culling the front-facing faces (with respect to the light) produces the identical effect of back-face culling. The front-facing version, however, culls faces from which shadow-rays are "launched" and therefore largely addresses the "shadow acne" problem. The shadow acne problem is an artifact of floating-point imprecision that causes the ray to intersect the surface from which it was launched. Front-face culling is similar to "second depth testing" as described by Wang [73, 77]. Additionally, rays that would originate on a back-facing face may be culled immediately without any tracing overhead.

5.2.2 Perspective Space

The specific transform I use for ray specialization is the perspective transform. The perspective transform is a non-affine transform that maps lines to lines (technically it is a projective transform). The perspective transform has the interesting property that rays sharing a common origin rays (hence-forth referred to a common-origin rays) in world space are parallel (and axis-aligned) in perspective space. Additionally, near-common-origin rays are mapped to nearly parallel (nearly axis-aligned) rays.

The perspective transform can be simply described using the following three formulas:

$$x' = x/z \tag{5.1}$$

$$y' = y/z \tag{5.2}$$

$$z' = -1/z \tag{5.3}$$

Alternatively we could use $z' = 1/z$. I use the presented formula because it preserves both ordering in z and the handedness of the coordinate system. Handedness can be important when implementing back-face culling.

Building and traversing acceleration structures in perspective space is no more complicated than building and traversing acceleration structures in world space. To use the perspective space transform to accelerate near common-origin ray tracing, one must simply transform all rays and geometry into perspective space (using the provided formulas) and use the same

structures and algorithms that would be used in world space. In short, using the perspective transform to build specialized acceleration structures is conceptually no more complicated than using an object space transform for object-specialization. Geometry must be transformed into the proper space before building the structure and rays must be transformed into the proper space before being traversed. The benefit in the case of perspective space is the alignment of the rays (rather than the geometry) with the coordinate axes of the acceleration structure.

It should be noted that ray specialization and object specialization are fundamentally at odds with one another. Aligning a single axis-aligned structure with a specific object and a specific collection of rays is an over-constrained problem. Therefore, for each collection of objects or rays, a specific alignment must be chosen. Ray alignment provides better performance than object alignment for near common-origin rays. It should be noted that all results in the implementation section comparing perspective space structures to world space structures align the coordinate axes to the dominate scene axes in world space. If this were not the case, the perspective aligned structures would perform relatively better.

5.2.2.1 The Perspective Singularity

Although the perspective transform has many useful properties such as mapping lines to lines, it has one challenging property that must be addressed. The perspective transform divides by z and thus has a singularity at $z = 0$.

This problem is well known in raster-graphics systems and a standard solution exists: clipping. Perspective space may be clipped such that no points lie on $z = 0$. The clipping methods commonly used are to clip to a near-plane (everything behind a plane close to the camera is culled) or to clip the world to the camera's view frustum.

It should be noted that because of the perspective singularity, perspective structures only deal with half (or otherwise partial) spaces. To use perspective space structures for lights or cameras with viewing angles greater than 180 degrees, it is necessary to use multiple different perspective space structures. A solution to this problem with six back-to-back structures is known as cube mapping. A practical concern with respect to acceleration structures is that this increases the number of acceleration structures by a factor of six. However, this increase in the number of acceleration structures shouldn't increase the overall build cost, because very little geometry is duplicated between the structures.

The implementation and results section of this dissertation contains the details about my implementation of ray-specialized acceleration structures, including the perspective grid and the perspective kd-tree as well, as comparisons to related work.

5.3 Fast Build

Specialized acceleration structures, by their nature, are not typically used for general visibility queries. Therefore a system that uses specialized ac-

celeration structure must typically use many of them. These numerous acceleration structures need to be constructed and maintained. This section describes two methods for building high-quality acceleration structures quickly. As mentioned previously, many specialized acceleration structures don't need to be rebuilt. Structures with static topologies only require a simple refit. Static structures don't need to be modified at all. However, fully dynamic structures such as those used to manage characters or particles with arbitrary movement may need to be rebuilt every frame. I present two algorithms for improving rebuild SAH based kd-trees that have the net effect of reducing build time by two orders of magnitude when compared to the traditional sort based SAH build while only sacrificing a very small amount of rendering performance.

5.4 SAM Scan

In this section I present an algorithm for quickly choosing kd-tree split planes that are close to optimal with respect to the SAM criteria. The approach approximates the SAH cost function across the spatial domain with a piecewise quadratic function with bounded error and picks minima from this approximation. The algorithm takes full advantage of SIMD operations (e.g. SSE) and has favorable memory access patterns. In practice the algorithm is faster than sorting-based SAH build algorithms with the same asymptotic time complexity and is competitive with non-SAH build algorithms which produce lower-quality trees. The resulting trees are almost as good as those produced by a sorting-based SAH builder as measured by ray tracing time. For a test

scene with 180k polygons, the system builds a high-quality kd-tree in 0.26 seconds (on a 2.66Ghz Core2) that only degrades ray tracing time by 3.6% when compared to a full-quality tree. For many applications, a rendering system must perform at real-time frame rates and support fully dynamic scenes. If optimized kd-tree construction were sufficiently fast, fast kd-tree-based ray tracing algorithms such as MLRTA [58] could be leveraged for these applications. The algorithm assumes the kd-tree is constructed from a "soup" of axis-aligned bounding boxes (AABBs). Each AABB may contain one or more triangles or any other desired leaf geometry.

The analysis and implementation focus on kd-trees, and the most immediate contribution is the demonstration that SAH-optimized kd-trees are a viable acceleration structure for dynamic scenes. However, the technique described here is directly applicable to the construction of other high-quality acceleration structures. Wald [67] extends these ideas to build bounding volume hierarchies. The most important conclusion drawn from this algorithm is that acceleration structures for interactive ray tracers can and should be built using a good approximation to the SAH.

5.4.1 Evaluating the Cost Function: Sorting vs. Scanning

Acceleration structure build algorithms use the surface area metric to choose a low-cost scene partition. To accomplish this task, the algorithm typically evaluates the cost function for several candidate partitions. In the case of a kd-tree this partition is defined by a split plane. To evaluate the cost

function for a particular plane we must know:

1. The location of the split plane (which allows us to compute the surface areas of the left and right nodes, and thus the probability of hitting them)
2. The number of primitives to the left of the split plane
3. The number of primitives to the right of the split plane

Tasks 2 and 3 are expensive. There are two basic algorithms for determining these values. We will assume m primitives (assumed to be axis-aligned bounding boxes (AABBs)) at a particular node and that we wish to evaluate the cost function at q different locations along a single axis of that node.

Sorting Approach: The sorting approach consists of two phases. In the first phase, the primitives are sorted along the axis, at a cost of $O(m \log m)$ for m primitives. In the second phase, the cost function can be evaluated for any number of desired locations with only a single pass over the sorted data at a cost of $O(m)$. For the kd-tree as a whole, the cost of this approach is $O(n \log^2 n)$ for n total primitives. However, by preserving and reusing the results of the top-level $O(n \log n)$ sort, the total cost can be reduced to $O(n \log n)$ [69].

Scanning Approach: The scanning approach evaluates the cost function at just a single location. In its simplest form, the algorithm must be repeated to evaluate the cost function at more than one location. For each

location, the algorithm loops over all of the primitives and tests each primitive to determine if it lies below and/or above the location and increments the appropriate counter(s). Thus to determine the number of objects on each side of a location, this operation requires $O(m)$ work. For q locations, the cost is $O(qm)$. If q is a constant (e.g. eight), then the overall work is still $O(m)$ to evaluate a set of split candidates. With an $O(m)$ cost per node, the total cost of producing an acceleration structure is $O(n \log n)$.

From asymptotic analysis, the costs of the two approaches appear to be equivalent. Additionally, since the sorting approach evaluates the cost function at more places, it would initially appear to be the better approach. However, there are several practical advantages of the scanning approach. First, it is very simple and thus the constant factors in its cost can be very small, especially when the implementation is well-tuned. Second, the scanning approach defers more work to the leaf nodes. Deferring work to the leaf nodes has two advantages. First it may be the case that we do not need to build all of the leaf nodes. If a system builds the acceleration structure lazily (e.g. the systems described by Lauterbach [48] and Djeu [19]) then it can skip a large amount of work. In contrast, the sorting approach still requires $O(n \log n)$ work at the top level of the hierarchy. Additionally, the scanning approach performs well at higher levels of the hierarchy where the data set does not fit into cache and the sorting approach is less efficient. I present substantially improved performance when using a scan based builder in conjunction with other optimizations such as lazy build and build from hierarchy (described in

the next section).

5.4.2 Approximating the Cost Function With a Few Samples

The scanning approach to evaluating the cost function outperforms the sorting approach if the number of locations at which the cost function is evaluated is small. Fortunately, a small number of locations is generally sufficient to build a high-quality kd-tree. For example, Hurley et al. [39] found that there was very little benefit to using more than 32 candidate split locations.

The SAH traditionally only chooses split planes lying on the locations at which the cost function as been evaluated [25]. I show that it is possible to use a small number of evaluations of the cost function to generate an approximation of the true cost function. The final split plane is then positioned at the minimum of the approximated cost function. Thus the location of the final split plane is not restricted to a fixed number of locations. Figure 5.1 illustrates this idea.

A good approximation to the surface area metric must meet two criteria. First it must significantly reduce the time required to build the acceleration structure. I assess this criterion using execution time measurements for the algorithm. Second, it should not significantly reduce the quality of the acceleration structure.

I assess the quality of the acceleration structure in three ways: first I measure the increase in ray tracing time that results from using our approximate tree instead of one built using the fully-accurate SAM. Second, I measure

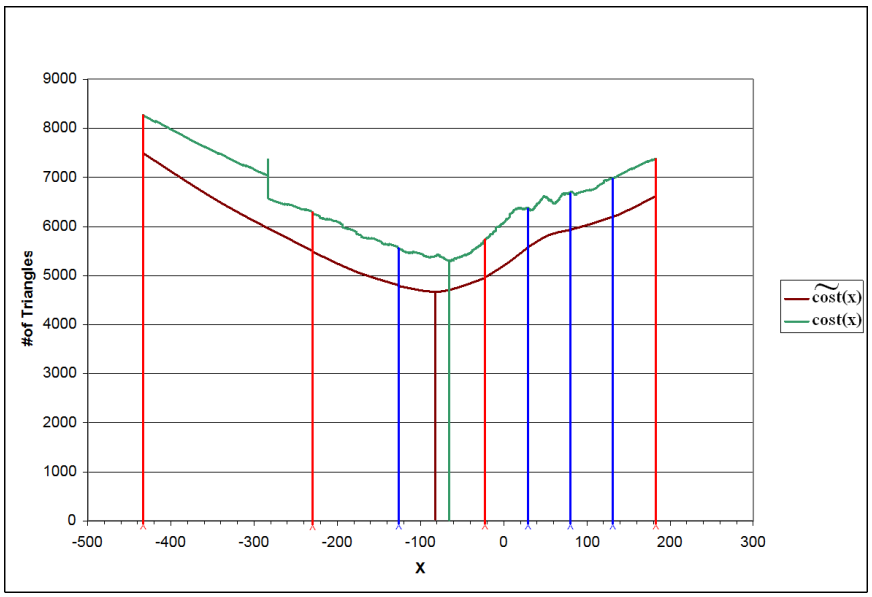

Figure 5.1: The split plane is placed at the minimum of a piecewise quadratic function that interpolates the sample points. Note that we have vertically displaced the approximation function from the actual function so that the details of both can be seen. The upper curve is the actual cost function. The lower curve is the quadratic approximation. Vertical lines are sample points.

the degradation in tree quality according to the SAM. Finally, I address the tree quality mathematically, by analyzing the behavior of the SAM and deriving analytical bounds on the error that results from evaluating it at a fixed number of locations.

5.4.3 Adaptively Choosing Sample Locations

Previous approaches that approximate the cost function have chosen uniformly-spaced samples (shown in Figure 5.2). Here we show that it is possible to achieve a better approximation to the cost function by adaptively choosing sample locations. This approach has been independently suggested (but not implemented) by Popov et al. [55].

Figures 5.2 - 5.4 illustrate how we choose samples adaptively. In the first phase 50% of our sample budget is used to uniformly sample the function. In the second phase the remaining 50% of our sample budget is used to adaptively sample the function in locations where there is the greatest uncertainty about the behavior of the function.

5.4.3.1 Error Bounds

For the purpose of discussing error bounds, it is useful to distinguish between the *greedy* SAM cost and the *true* SAM cost. The true cost recursively considers cost at all child nodes but can only be evaluated for a node after its entire subtree has been constructed. In contrast, the greedy SAM cost does not use recursion to evaluate the cost and ignores the effects of deeper tree

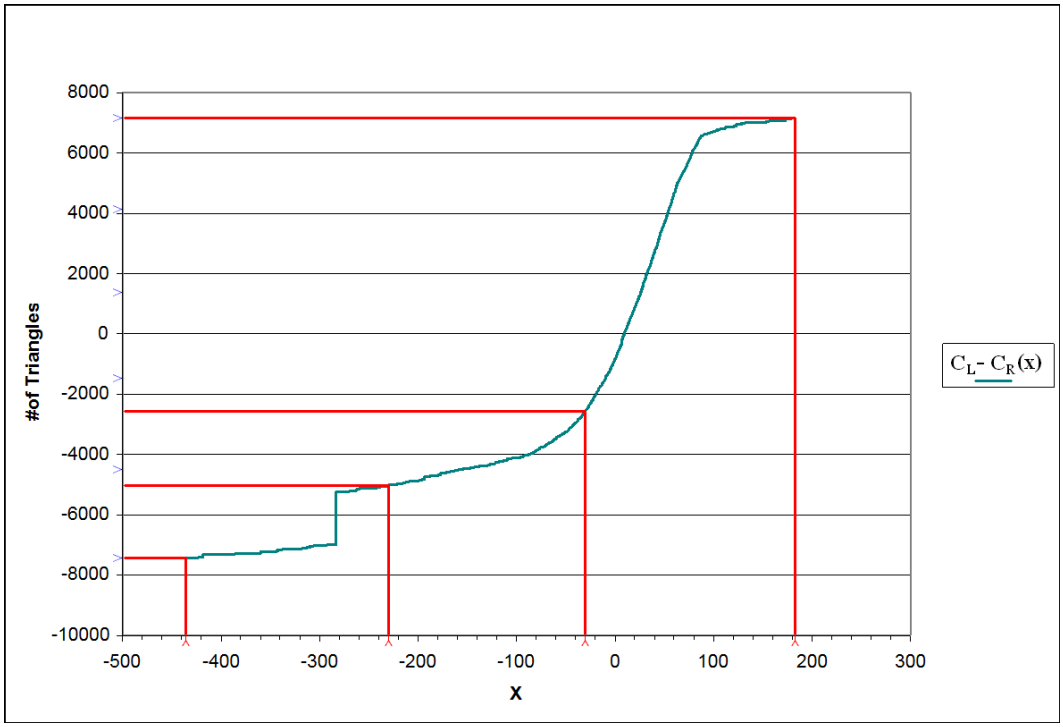

Figure 5.2: The initial samples as shown on $C_L - C_R$

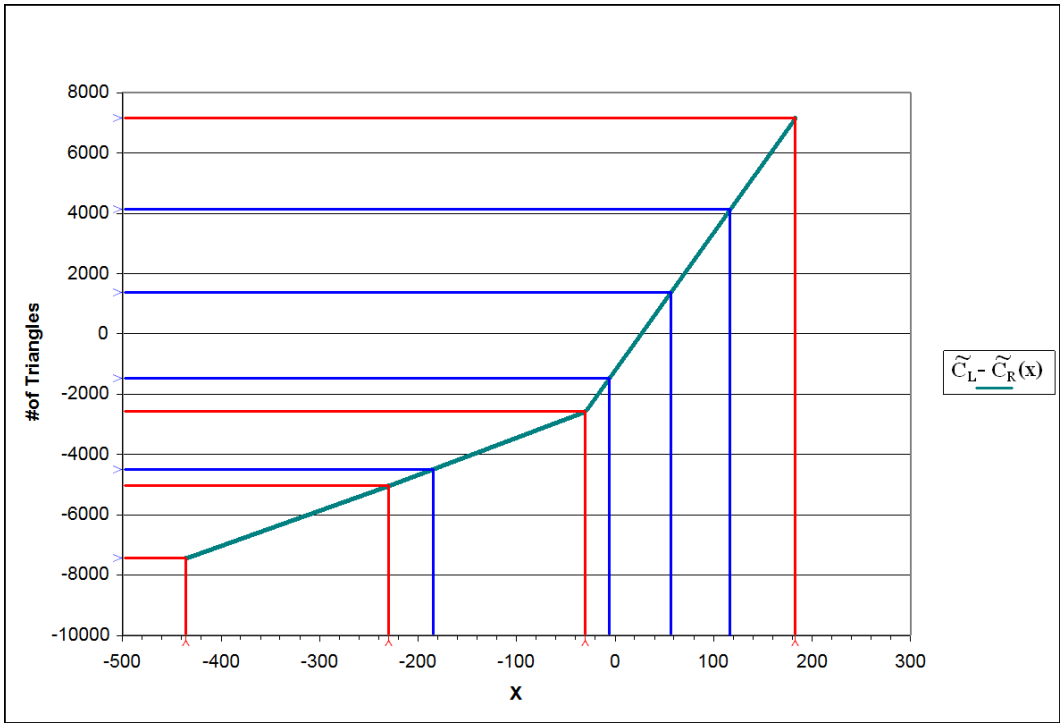

Figure 5.3: Use a comb sample along *range* of $C_L - C_R$ to find segments with large error.

structure.

It should be clear that the scan approximation is very effective in cases where the greedy cost function is smoothly varying as is typical near the top of the acceleration structure. In such smoothly-varying cases, the approximate cost function used by the algorithm closely matches the greedy cost function at all points. Thus, the greedy cost of the split plane chosen by the algorithm is very close to the greedy cost of the optimal split plane.

It is less obvious how well the algorithm performs in cases where the greedy cost function has discontinuities. For example, if a portion of the scene

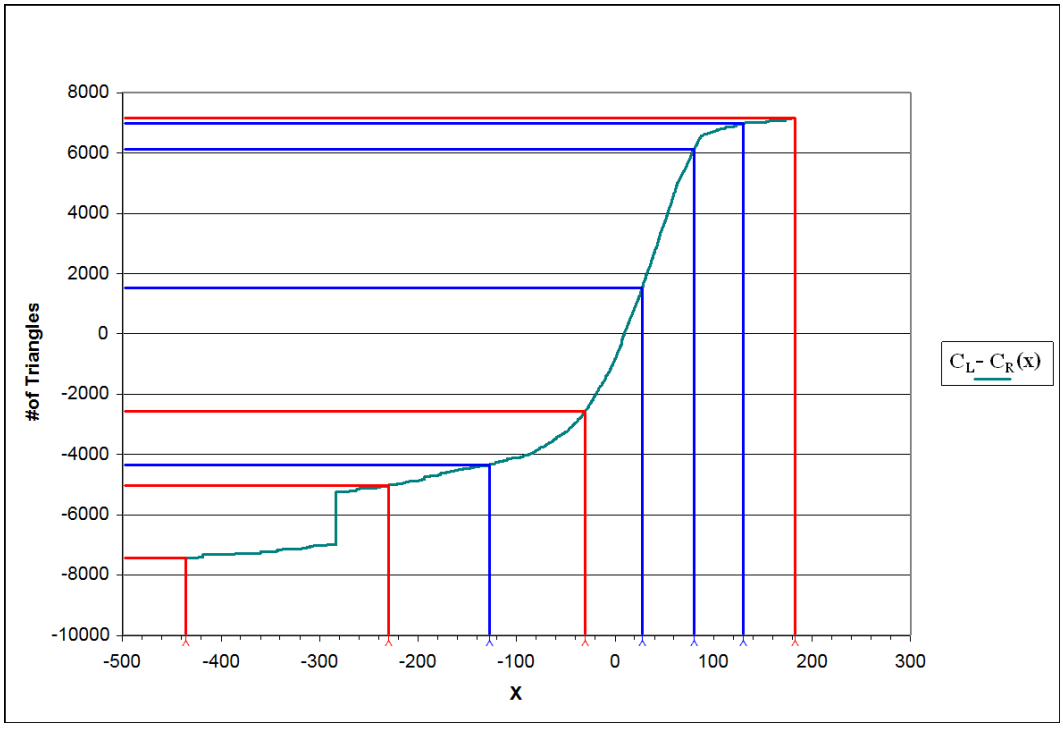

Figure 5.4: Sample evenly in the segments with large error.

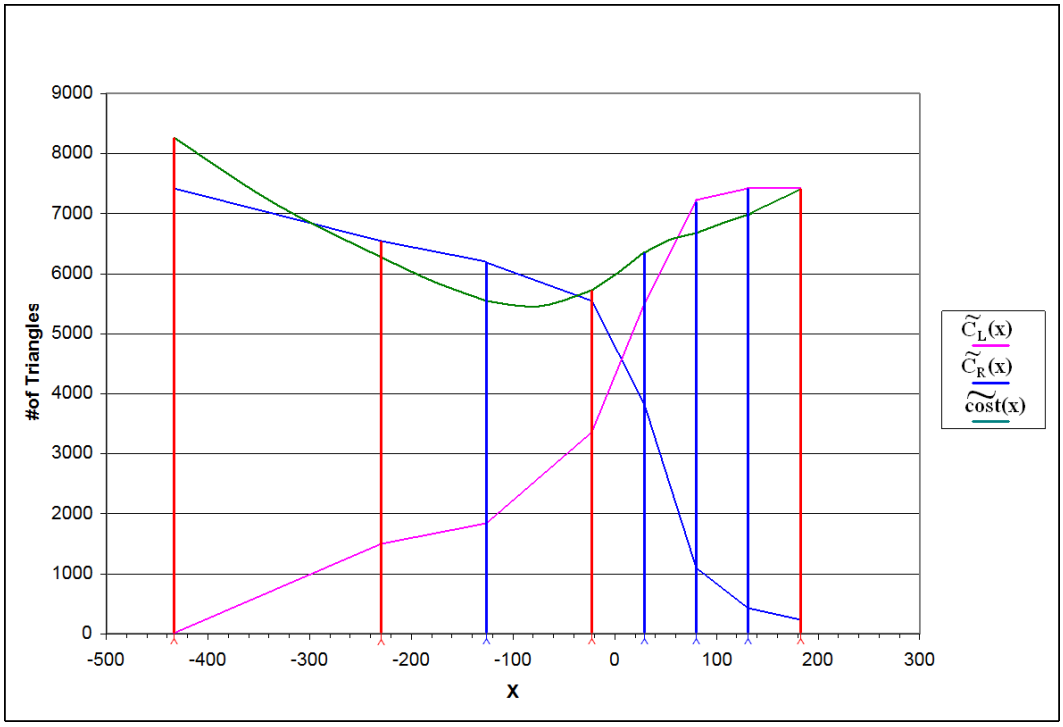

Figure 5.5: Approximate C_L, C_R and *cost*.

contains an axis-aligned wall with many polygons, the cost function can have a large discontinuity. This effect is visible in the example diagrams. In such cases, the greedy cost of the split plane chosen by the approximation may differ significantly from the greedy cost of the optimal split plane (specifically by as much as half of the range of the cost function). A step function has a large high frequency component (up to infinite) so any discrete sampling strategy suffers this deficiency.

However, it turns out that in such cases the *greedy* SAM cost does not correspond well to the *true* SAM cost if the next child splits are chosen well.

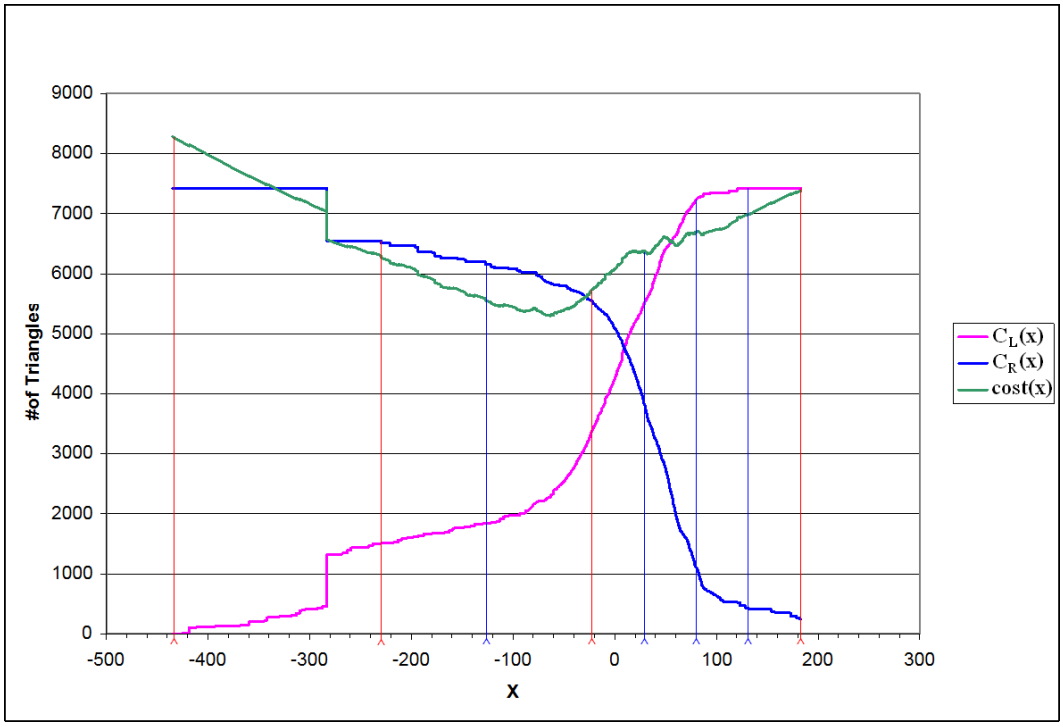

Figure 5.6: The actual cost function, notice the minimum is slightly off from the predicted value.

The algorithm chooses the child splits well for reasons that I will explain next. As a result, the degradation in the *true* SAM cost caused by the approximation is typically small even for a cost function with discontinuities.

A key property of the algorithm is that the samples chosen rapidly converge towards a discontinuity. This property is due to the fact that adaptive samples are placed in locations of high error. Discontinuities produce regions of high error in the approximation and attract adaptive samples. Between adaptive samples and the recursive splitting process, discontinuities are isolated quickly. In fact, the larger the discontinuity is, the more rapid the convergence (more adaptive samples near the discontinuity). The net effect of this convergence is that after a few splits the discontinuity is isolated into a small volume. Since the true SAM cost recursively considers splits and the algorithm converges rapidly, the true SAM cost penalty for non-optimal split locations near a discontinuity is generally small, even when the greedy SAM cost penalty for the approximate split is large.

Mathematically, the convergence property of the algorithm is expressed as a guaranteed bound on the product of the cost error with the split-plane position error. Specifically, if there is a large bound on the error in the cost, there is a small bound on the error of the location of the split plane and vice-versa. This result falls out directly from the fact that the proof provides error bounds in terms of the product of the domain and the range of the function being approximated. Thus, for very large bounds on the cost error, as occur near a discontinuity in the cost function, the algorithm will choose a split plane

close to the discontinuity. After a few splits the discontinuity will be isolated into a small volume and thus contribute very little to the *true* SAM cost.

5.4.3.2 The Adaptive Sampling Algorithm

As discussed, the algorithm approximates the SAM by sampling it at a fixed number of locations. Instead of sampling the cost function itself, we sample all four varying inputs to the cost function (C_L, C_R, SA_L, and SA_R). By linearly interpolating each input, we are able to generate a quadratic approximation to the cost function between each pair of sample points.

It would be possible to stop after this step and choose a split plane at the minimum of the piecewise-quadratic approximation to the cost function. However, if the cost function is ill-behaved, the error bounds on the segment(s) that potentially contains the minimum may be loose. That is, there may be considerable uncertainty as to the actual behavior of the cost function in the vicinity of the minimum and thus the choice of the location of the minimum may be poor.

Figure 5.2 illustrates this situation. Note that in the referenced figure, rather than plotting the cost function we plot $C_L - C_R$. This function (as explained later) is useful as a tool for error analysis due to the fact that bounds on $C_L - C_R$ simultaneously bound C_L and C_R, which in turn bounds the cost function. In Figure 5.2 there is a large amount of uncertainty about the behavior of $C_L - C_R$ in the last segment.

To improve the error bounds, a second set of q samples is taken, with

the locations for the second set of samples chosen adaptively based on the information from the first set. The sample locations are chosen using a simple algorithm illustrated graphically in Figures 5.2–5.6. The algorithm places additional samples in those segments for which there is a large change in $C_L - C_R$ within the segment. A simple but effective mechanism for choosing these new sample locations is to create n bins over the range $C_L - C_R$ and then place an additional sample within each segment for each time that $C_L - C_R$ crosses a bin boundary within the segment. This process is equivalent to taking q additional samples regularly along the *range* of $C_L - C_R$ (illustrated in Figure 5.3).

Once it has been decided how many of the additional samples are allocated to each of the original segments, these additional samples are positioned such that they are evenly spaced within their respective segments (Figure 5.4). The end result of the two sampling steps is a piecewise quadratic approximation to the cost function using $2q$ samples. As shown in the next subsection, the product of the cost function and the position error has a $(1/q^2)$ bound between adjacent pairs of samples within this adaptive sampling technique. This bound is important because it governs the rate of convergence for iterations of the sampling process. In this context, scans within children may be considered iterations. To choose the split plane location for the node, the algorithm considers the local minima of each of the $2q-1$ piecewise quadratic segments and places the split plane at the overall minimum. As with any kd-tree builder, if the estimated cost of splitting at the location is greater than the cost of not

splitting, then no split is made and a leaf node is formed.

Implementation and results from this algorithm are discussed in the results section near the end of this dissertation. I have not demonstrated the practical benefit or necessity of using the quadratic interpolation of the surface area metric samples instead of a linear one, but it adds only a small constant amount of additional work and is certainly required to guarantee the provided error bounds.

5.4.4 Error Bounds

In order to bound the error of the scan approximation to the SAH cost function, I first bound the error of each of its components. The focus of most of this subsection is to provide error bounds for the piecewise linear approximations of the monotone functions C_L and C_R.

5.4.4.1 Linear Approximation

For an arbitrary integrable function f and an approximation $\widetilde{f}$, we may define the error of the approximation over domain (a, b) to be $\int_a^b |f - \widetilde{f}|$. If f is a monotone function and $\widetilde{f}$ is a linear approximation of f over (a, b) with $f(a) = \widetilde{f}(a)$ and $f(b) = \widetilde{f}(b)$, the maximum error of the approximation is $\frac{|b-a||f(b)-f(a)|}{2}$. I do not provide proof of this simple theorem here, but Figure 5.7 should provide some intuition as to why it is true. I refer to this bound as the linear approximation bound (of a monotonic function).

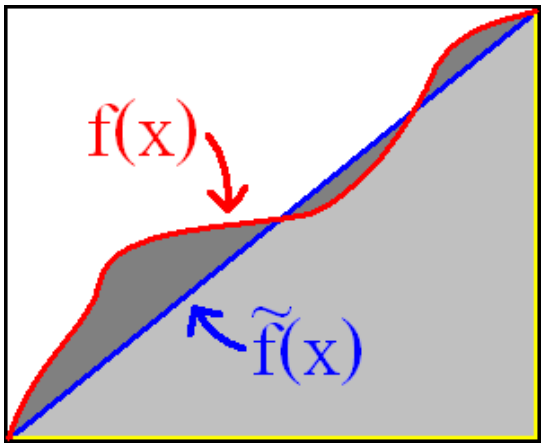

Figure 5.7: Monotonicity guarantees that the function does not leave the box. The area between the function and its linear approximation cannot be more than $\frac{1}{2}$ the area of the box.

5.4.4.2 Uniformly Spaced Linear Approximation

A uniformly spaced piecewise linear approximation provides a better error bound for monotone functions. When approximating a monotone function f, we may take n evenly spaced points $x_i, i \in [0, n)$ **s.t.** $x_0 = a$ and $x_{n-1} = b$. Let $\widetilde{f}$ be the piecewise linear approximation of f such that $\forall x \in [0, n) f(x) = \widetilde{f}(x)$ and $\widetilde{f}$ is linear between all x_i, x_{i+1} pairs. The following bound holds given the assumptions: f is an integrable monotone function,

$n \geq 2$ and (a, b) is a non-empty range.

$$
\begin{aligned}
error(\widetilde{f}) &= \int_a^b |f - \widetilde{f}| \\
&= \sum_{i=1}^{n} \int_{x_{i-1}}^{x_i} |f - \widetilde{f}| \\
&\quad \{\text{the linear approximation bound}\} \\
&\leq \sum_{i=1}^{n} \frac{|x_i - x_{i-1}||f(x_i) - f(x_{i-1})|}{2} \\
&\quad \{\text{f is monotone}\} \\
&\leq |\sum_{i=1}^{n} \frac{|x_i - x_{i-1}|(f(x_i) - f(x_{i-1}))}{2}| \\
&\quad \{x_i - x_{i-1} = \frac{b-a}{n-1}\} \\
&= \frac{(b-a)}{2(n-1)} |\sum_{i=1}^{n} (f(x_i) - f(x_{i-1}))| \\
&\quad \{f(x_i)\text{is a telescoping sum}\} \\
&= \frac{(b-a)|f(b) - f(a)|}{2(n-1)}
\end{aligned}
$$

This bound improves over the linear approximation bound by a factor proportional to $\frac{1}{n}$. I will refer to it as the evenly spaced piecewise linear approximation bound (of a monotone function).

5.4.4.3 Adaptive Linear Approximation

Although the evenly spaced piecewise linear approximation provides an improved error bound for $\widetilde{f}$, it provides no information about the locality of error. Each segment (x_i, x_{i+1}) contains some fraction α_i of the overall error

in our piecewise linear approximation but we have no guarantee that any one segment doesn't contain most or all of the error. By adding at most n additional samples (using my adaptive method), we may ensure that each segment contains proportional to $\frac{1}{n}$ of the overall error.

Using the piecewise linear approximation bounds, adding $\lfloor n\alpha_i \rfloor$ evenly spaced points to a segment reduces its error proportional to $(\lfloor n\alpha_i \rfloor + 1)^{-1}$ (using the evenly spaced linear approximation over with that segment). It should be noted that two additional samples are available in each segment: the endpoints. These additional samples are responsible for the addition, rather than the subtraction, of 1 in this reduction of error. Error for the i^{th} segment was initially $\alpha_i \frac{(x_{i+1}-x_i)|f(x_{i+1})-f(x_i)|}{2(n-1)}$. After the additional samples it becomes $\alpha_i \frac{(x_{i+1}-x_i)|f(x_{i+1})-f(x_i)|}{2(n-1)(\lfloor n\alpha_i \rfloor+1)}$. This quantity is proportional to $\frac{(x_{i+1}-x_i)|f(x_{i+1})-f(x_i)|}{2n^2}$ and provides us with an error bound on each segment proportional to $\frac{1}{n^2}$. I refer to this bound combined with the evenly spaced linear approximation bound as the adaptive linear approximation bound. The inequality $\sum_{i=1}^{n} \lfloor n\alpha_i \rfloor \leq \sum_{i=1}^{n} n\alpha_i = n$ demonstrates that this process requires at most n additional samples to guarantee the error bound.

5.4.4.4 Bounds for the Scan-Based Cost Function

Recall the SAH cost function:

$cost(x) = C_I + C_L P_L(x) + C_R P_R(x)$

Because C_L and C_R are monotone, we may sample them adaptively and achieve the adaptive linear approximation bound. The quantities C_I, $P_L(x)$

and $P_R(x)$ are computable directly and have no error. The error bounds for the cost function are derived here.

$$\begin{aligned} error(cost(x)) &= error(C_I) + error(C_L)P_L(x) + error(C_R)P_R(x) \\ &\quad \{P_L, P_R \in [0,1], error(C_I) = 0\} \\ &\leq error(C_L) + error(C_R) \end{aligned}$$

Because C_L and $-C_R$ are both monotone increasing, so is their sum. Also, since the range of $C_L - C_R$ is equal to the sum of the ranges of C_L and C_R, any bound for $C_L - C_R$ is also a bound for C_L and for C_R. Using the adaptive bound for $C_L - C_R$ we can show that the error between any two samples over $C_L - C_R$ is of the order $\frac{(b-a)(|C_L(b)-C_L(a)|+|C_R(b)-C_R(a)|)}{n^2}$. The same bound holds for the error between any two samples on C_L and $-C_R$. Adding the error bounds for C_L and $-C_R$ together increases overall error by a factor of two and does not change the order of the error. The quantities P_L and P_R are ≤ 1 and multiplication with them does not change the order of the error either. Therefore, the error between any two points in $cost(x)$ is also order of $\frac{(b-a)(|C_L(b)-C_L(a)|+|C_R(b)-C_R(a)|)}{n^2}$.

5.4.5 Comparison to Related Work

A similar SAM scanning technique was developed concurrently by Popov [55]. While both algorithms improve kd-tree build performance by approximating

the SAM in a machine-friendly way, several major differences exist between the two algorithms. In this subsection I will break the algorithms apart and analyze the differences between them, including the motivations that lead to these differences.

5.4.5.1 Streaming Construction

Both approaches use a streaming construction approach. The algorithms stream geometry across a collection of previously selected candidate split locations. Streaming greatly improves memory performance and is one of the key reasons for the high performance of both approaches. In addition to streaming geometry, Popov's paper builds kd-trees in breadth-first order at the top of the tree and switches to depth-first near the leaves. This breadth-first order enables globally linear memory access when choosing multiple splits. My system builds in depth first order.

5.4.5.2 Sampling the Cost Function

The two approaches use a significantly different sampling methodology. As described above, my approach takes two sets of relatively few (8) samples and the second set is adaptively chosen. Every sample is tested against every object during my scan (this is inexpensive due to the use of SIMD and the fact that the samples stay in registers). Popov's paper takes relatively many samples (1024). These samples are always uniformly spaced in order to avoid having to test each sample individually against an object. The bounds of the

object are used to directly calculate the index of the first and last samples affected by the object. Counters for each sample only store the number of objects that begin or end near that sample. The left and right counts for each sample are reconstructed via a final scan and accumulate over all of the samples.

5.4.5.3 Reconstruction

The two papers also use different reconstruction approaches. My algorithm linearly interpolates the left and right counts and exactly evaluates the probabilities, resulting in a quadratic interpolation of the final cost function. Popov's approach linearly interpolates the cost function directly.

5.4.5.4 Specialization at the Leaves

Each algorithm also handles lower levels of the kd-tree differently than it does higher levels. My approach uses a machine friendly brute force $O(n^2)$ scanning algorithm to evaluate leaves. Popov's approach uses a machine friendly radix sort in a traditional sort-based kd-tree building approach at the leaves.

5.4.5.5 Overall Results

Overall, my implementation of the scan algorithm is approximately twice as fast as Popov's implementation for his algorithm for a given scene. The speed advantage is likely mostly due to the extremely tight inner loop in

my scan approach (only one read from memory and no writes). Also, the brute force scan is likely to be faster than any sort for very small nodes. Overall however, the algorithms are based on similar principles and have a similar overall performance.

5.5 Build From Hierarchy

In the previous section, I described an algorithm for approximating the surface area metric in order to improve kd-tree build performance. In this section, I show how to use structural information about a scene such as the information contained in a scene graph to build SAM-based acceleration structures more efficiently. I present a build algorithm that uses structural information to build acceleration structures more quickly. I also provide asymptotic analyses for both standard and lazy variants of my build algorithm. In particular I show bounds of $O(n)$ work for full kd-tree builds over n primitives and $O(v + \log n)$ for lazy kd-tree builds over v visible primitives. In the implementation and results chapter I provide experimental results showing that these asymptotic properties translate into practical speedups. I also show that under certain (realistic) assumptions on the scene structure, the method produces provably good acceleration structures. Finally, I provide experimental results demonstrating that the acceleration structures have nearly indistinguishable quality to those produced using a traditional SAH build.

Most existing SAH-based tree construction algorithms assume that input data is essentially a "soup" of polygons or axis-aligned bounding boxes

(AABBs). No global scene structure is assumed or taken advantage of in building the tree. However, real scenes have a great deal of structure. This structure is generally encoded as a scene graph representing the natural boundaries between and the relationships among the objects in a scene. A scene graph is a hierarchical collection of objects. Rigid body motions in the scene are represented by transformations between local coordinate frames associated with the objects represented in the scene. Natural clusters of geometry in the scene are found within objects. Thus, the information available in scene graphs is quite pertinent to the heuristics used for ray tracing acceleration.

While some batch ray tracers (e.g. Pixar's PRMan) have exploited scene graph information, the Razor interactive ray tracer [19] is the first to my knowledge to exploit such information in an interactive or real-time system. Razor incorporates a fast SAH-based kd-tree build scheme based on an input scene graph and the use of the scan-based surface area metric approximation scheme presented in the previous section [38]. Use of an input scene graph enables Razor to use a lazy build scheme to further accelerate the kd-tree builder.

5.5.1 Building Acceleration Structures for Dynamic Scenes

In batch ray tracing systems, acceleration structure construction is often considered to be a "free" preprocessing step. However, as interactive ray tracing has become practical, it has become important to consider fast techniques for building and updating acceleration structures.

For scenes containing deformable motion, three broad strategies have recently emerged [71]. The first strategy is to choose an acceleration structure such as a regular grid that is especially simple to build [70]. The second strategy is to use high-performance algorithms to build a high-quality structure such as a metric-based kd-tree [19] or BVH [67]. The third strategy is to incrementally update the acceleration structure each frame [48, 79]. We focus on the second strategy in this dissertation. The analysis assumes a preexisting hierarchy as the starting point for the build. This starting point can come from an input scene graph or a previous frame's acceleration structure. In the latter case the distinction between rebuilding and updating the acceleration structure becomes blurry.

5.5.2 Using Hierarchy to Accelerate Acceleration Structure Build

The traditional SAH build algorithm assumes that the structure is being built "from scratch." Good acceleration structures perform two roles in accelerating ray tracing. First, they help eliminate wasted work testing for intersections between objects and rays that are not close by. Second, they allow the ordering of intersection tests along a ray. This ordering essentially amounts to sorting the scene geometry and accounts for the $\Theta(n \log n)$ complexity of acceleration structure build. However in practice, a great deal of structural information about the scene geometry is usually known at the time of the structure build. Such information could be obtained from a scene graph or a previously-constructed acceleration structure. When this structural in-

formation is sufficient to amount to a "pre-sort" of the scene geometry, using it can allow a new acceleration structure to be built in linear time rather than $O(n \log n)$ time.

I now sketch the approach for building an acceleration structure over n primitives in $O(n)$ time. Linear time complexity is achieved by doing constant work to determine each split plane during acceleration structure build. Constant time work per node is achieved by only examining a constant-size subset of scene objects when determining each split plane. Sorting a constant-size set takes constant time guaranteeing constant work per node. The algorithm obtains this constant number of objects via careful refinement of the provided scene-graph. The approach is also guaranteed (under reasonable assumptions) to produce a "good" acceleration structure. The primary assumption required is that each of these constant-size subsets of objects has a normal enough spatial distribution that a "good" acceleration structure exists for it (the details of this assumption are covered in a later subsection). In this case I show that the algorithm will produce a structure of quality comparable to the quality of a structure built using a full build. Thus with an appropriate input structure, my algorithm will produce an acceleration structure guaranteed to be high-quality and do so in linear time.

In the remainder of this section I will formalize the algorithm and the intuitions I have provided. A detailed discussion of the definition of a "good" acceleration structure is presented at the end of this section and in my prior publication [37]. First I will define more precisely the structural requirements

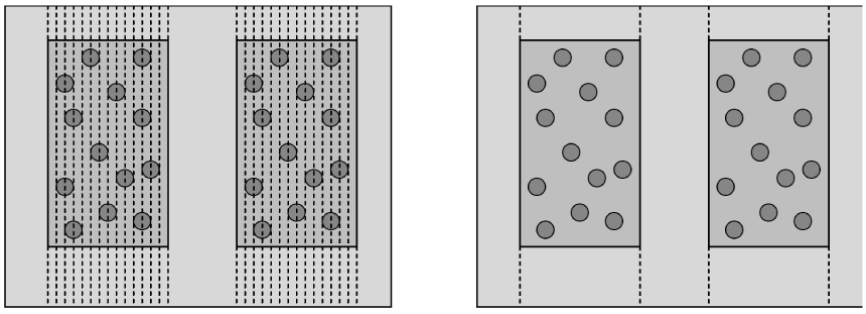

Figure 5.8: The use of hierarchy can greatly reduce the number of split candidates provided to a builder. If the hierarchy fits the geometry well, then the provided splits are often among the best anyway. Vertical dashed lines in these diagrams represent split candidates.

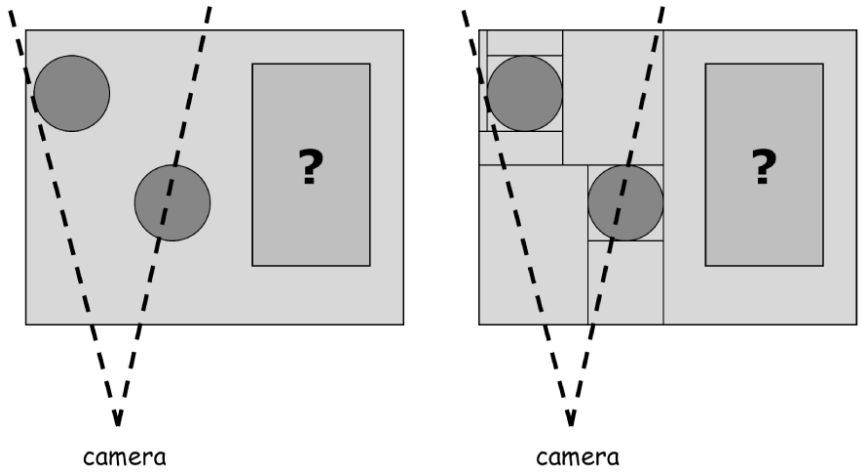

Figure 5.9: Using hierarchy in conjunction with lazy build can allow a builder to completely ignore large regions of space if no rays enter that space.

on the input that will allow us to produce good acceleration structures on output. I will show a bit more formally why this process takes linear time. I will then examine various strategies for enhancing this basic scheme in order to provide better structure quality and safeguard against failed assumptions. Finally, I provide experimental confirmation of our formal analyses.

5.5.3 Building Acceleration Structures from Input Hierarchies

In this section I describe an algorithm for constructing a metric-based acceleration structure from an existing hierarchy. In a subsection, I demonstrate that the algorithm will produce a "good" acceleration structure according to a criterion I define and demonstrate practical results supporting this claim. I also show that, under reasonable assumptions, the algorithm works in $O(n)$ time for a scene with n primitives.

The algorithm is based on the traditional SAH build algorithm. At each step a split plane is chosen and present geometry is sifted into two bins based on their positions relative to the split plane. How this geometry is sifted can be different depending on whether we are constructing a kd-tree or a BVH. The algorithm uses groups of objects (represented by higher nodes in the scene graph) in place of individual objects in order to reduce the amount of geometry considered when choosing a split plane. We may pick a threshold a such that the algorithm only considers $O(a)$ objects when determining a split plane. In practice this number can be any number > 1. Smaller numbers result in faster build times but increase the dependence on the quality of

the input structure. If a very high-quality input structure is available and performance is a concern, a can be as low as 10. Fifty candidates seem sufficient in practice in most cases. In the Razor system, build takes such a small portion of the overall time that 500 is used to ensure high-quality output trees. The algorithm, using 500 objects for determining split planes, produces trees of virtually indistinguishable quality when compared to trees produced with the full sort in practice.

Note that the number of objects selected should be at least a if possible in order to have a sufficient number of split candidates from which to choose a good split plane. We can collect these objects by refining the input hierarchy nodes one at a time in a top-down order until we have at least a nodes. In some sense it doesn't matter how we choose the nodes to refine, but using a FIFO queue (breadth-first traversal) works well. If we have a binary hierarchy we can always obtain exactly a objects. Otherwise we will obtain at most $a+F_I-1 = O(a+F_I)$ where F_I is the maximum fan-out of the input hierarchy. The one exception to this rule is if there are simply not enough objects once all nodes have been refined.

Split planes are chosen according to the SAM as is normally done using the SAH. Split candidates are the bounding planes of the refined hierarchy nodes. The cost in the SAM is still estimated using the number of primitives overlapping each side of the split plane. To compute this estimate quickly, each input hierarchy node should know how much geometry it contains. If the SAM dictates that splitting the node produces a higher-cost tree than not

splitting the node, then my algorithm does not split the node (in the same manner that the automatic termination criterion works for the SAH.)

By reducing the amount of geometry considered to be a constant the work done at each node to determine each split plane is bounded to a constant (a significant improvement over previous approaches). It should again be noted that an acceleration structure created with this approach will not exactly match an acceleration structure created using with the traditional SAH. Different geometry leads to different decisions made by the SAH builder. It should also be noted that changing the input geometry can change the point at which the algorithm chooses to terminate the build process and produce a leaf. However, given a reasonable input hierarchy, the algorithm will produce good-quality results. I demonstrate high-quality results in the implementation section of this document.

5.5.4 Build Complexity

I now turn attention to the asymptotic analysis of the performance of an SAH build from an input hierarchy using my method. The analysis will require the assumption that the SAH will produce an $O(\log a + F_I)$ depth tree over any $\log a + F_I$ depth refinement ($\log a + F_I$ size subtree) from the input hierarchy. This assumption, which will guarantee that our algorithm will produce $O(n)$ tree nodes, is an extension of the assumption that the scene geometry has the property that the SAH will produce an $O(\log n)$ tree over it. The extension is a local criterion over each $a + F_I$ size subtree. The global

version of this assumption is also made in previous work in which an $O(n \log n)$ bound is derived [69] and holds true for non-contrived scenes in practice.

Observation: Given the assumption above, building an SAH based acceleration structure from a hierarchy is $O(n)$ for n geometric primitives.

Proof: Each node in the tree is built considering $x : x < a+F_I$ hierarchy nodes to determine the split plane. Sorting $O(a+F_I)$ candidates takes $O((a+F_I)\log(a+F_I)) = C$ time. Since the completed tree has $O(n)$ nodes by assumption and we did C work for each node, the total work for the entire tree is $CO(n) = O(n)$.

It should be noted that any SAH build is $\Omega(n)$ because we have to touch each object at least once in order to add it to the kd-tree. Therefore SAH build from hierarchy is $\Theta(n)$. Not surprisingly, a linear upper bound on build is tight.

5.5.5 Analysis of Lazy Build from Hierarchy

Laziness (demand driven evaluation) is an optimization that can eliminate large amounts of work in systems where only part of the input data is required to produce an output. In the context of ray tracing it can be leveraged to avoid a large amount of unnecessary work. In future systems with large scenes, only a small fraction of which are visible from a single viewpoint, the advantage of building acceleration structures for only the visible portion of a scene will grow. The Razor [19] system uses laziness as an optimization. It only builds visible (or nearly visible) portions of the acceleration structure

at each frame. The algorithm lazily builds the acceleration structure by interleaving acceleration structure build with ray traversal such that kd-tree nodes are only built once they are needed by a traversing ray. In terms of the input hierarchy, only volumes that a ray penetrates will be used in the acceleration structure build.

Before beginning the analysis of the build from hierarchy algorithm using the lazy optimization, it is important to understand the interaction between laziness and various acceleration structure build methods. Neither the traditional build algorithm nor the $O(n \log n)$ [69] algorithm can make use of laziness. Both algorithms require a costly $O(n \log n)$ sort up front. Scanning/binning approaches [38, 55] discussed earlier in this chapter reduce the upfront cost to $O(n)$ but still touch all of the geometry upfront. In fact any build (even median split) that doesn't use an input hierarchy will sift all n objects across the first split plane after choosing the first split. In order to build a truly SAH-based acceleration structure, an approach in which the initial split *and* sift are sub-linear in the number of nodes is necessary. The build from hierarchy approach does just that.

For the analysis of the lazy variant of the build from hierarchy algorithm, I will introduce a new variable $v : v < n$, the number of *visible* triangles. Laziness only provides a substantial advantage when $v \ll n$, so I will make that assumption for this analysis. Other assumptions will differ only slightly from the analysis in the previous (non-lazy) section. Primarily, I assume that the SAH will produce a $\log v$ depth tree over the set of *visible* primitives. This

assumption is essentially the same assumption as before, but restricted to the visible subset of the geometry. The implication of this assumption is that the SAH will produce a tree with $O(v)$ nodes.

Observation: Given the assumptions of this section and the previous sections, building an SAH-based acceleration structure using the lazy variant of build from hierarchy is $O(v + \log n)$

Proof: Each node in the tree considers $x : x < a + F_I$ hierarchy nodes when determining a split plane. Sorting $O(a + F_I)$ candidates takes $O((a + F_I) \log(a + F_I)) = C$ time. Since the completed tree is assumed to have $O(v)$ nodes and we did C work for each, the total work is $CO(v) = O(v)$. Additionally, we must have descended in our input hierarchy to a depth of $\log n$ in order to access any geometry that makes it into our kd-tree, resulting in a final bound of $O(v + \log n)$.

It should be noted that lazy SAH build is $\Omega(v + \log n)$. We still have to touch each visible object in order to add it to the tree and assume that we can only touch geometry through some sort of hierarchy or other $\log n$ depth structure. Therefore, lazy SAH build from hierarchy is $\Theta(v + \log n)$. As a practical matter, we expect $\log n \ll v$ implying that from a practical standpoint lazy build from hierarchy is linear in v.

5.5.6 The Effects of Fast Scan

Previous approaches to building high-performance acceleration structures, such as those presented earlier in this chapter, show that significant

improvements in performance can be made by approximating the SAM instead of computing it directly. These approaches combine effectively with build from hierarchy. Simply, we may scan our a candidate objects in order to find a minimum instead of sorting them. This change doesn't affect the asymptotic performance of the build algorithm (as noted above, the bound is already tight), but it does improve the constant factors. Specifically it improves the value C, the amount of work done to compute each split plane from $O((a+F_I)\log(a+F_I))$ to $O(a+F_I)$. This improvement can be quite noticeable if either a or F_I is relatively large. The scan provides a convenient "performance safety net" in the case that assumptions about fan-out are broken in a real world example. The experimental results presented in the implementation chapter show that the effects of the fast scan approximation combine nicely with both laziness and build from hierarchy.

5.5.7 High Quality SAM-Based Structures

The SAM has long been the most commonly used basis for heuristics used to build high-quality acceleration structures. The SAM is commonly used because it captures the expected cost of traversing a ray through an acceleration structure given the assumption that the ray direction was drawn from a uniform distribution in ray space. Minimizing the SAM cost of a structure minimizes the expected cost of traversing a ray through that structure. The SAH however, is a heuristic in the sense that it is not guaranteed to produce a structure that globally minimizes the SAM cost for an acceleration structure.

The global minimum is generally unknown for any given structure, because it is intractable to compute. Given the SAM, it is very simple to determine which of two trees has lower expected cost, but without a global minimum cost it is difficult to determine if a given tree is "good." When comparing trees built from hierarchy and trees produced by the traditional SAH, we require some notion of "good." In this section I define such a notion so that I can later show that the algorithm produces "good" trees.

When a ray reaches a leaf in an acceleration structure, it is tested against all of the primitives present in that leaf. To limit the work done at a leaf, we would like the leaves of a "good" acceleration structure to contain at most a constant c number of primitives. Not only should a good acceleration structure have limited size leaves, but the fan-out of every node in the hierarchy should also be bounded by a constant F. Finally, in a "good" structure, the bounding volumes of the children of any internal node should have at least some fraction p smaller surface areas than their parent. Therefore, for each child, the probability of a ray intersecting a child given that it intersected the parent is bounded. Not all scene geometry can guarantee these properties (thus permitting the existence of a "good" acceleration structure) but most do in practice.

I will now derive the cost of a "good" structure. The cost of intersecting a leaf node is at most c. Let the average conditional probability of intersecting a child be p. Using these two costs, we may compute the cost of the nodes with children that are leaves to be $c(Fp)$. Similarly, the cost of nodes with

grandchildren that are leaves is $c(Fp)(Fp)$. By induction, it is easy to show that the SAM cost of the root node is $C_{SAM} = c(Fp)^{log_F n} = cn^{1+\log_F p}$ for a structure over n primitives. Since $p < 1$ the log portion of this expression is negative, and the resulting equation is sub-linear in n.

It is unlikely that p is the same for all sets of children, and thus this derivation is only accurate in an ideal case. However, the SAM may be directly evaluated for a particular hierarchy. We can invert the formula from the previous paragraph to obtain a useful "average" value of p for the entire tree. We will refer to this "average" p as $q := (\frac{C_{SAM}}{cn})^{\frac{1}{\log_F n}}$. The new quantity q is an abstract measure of tree quality that is independent of n (in specific contrast to the C_{SAM} cost which is dependant on n). Even though we use n to compute the value of q, it should be noted that q is an average conditional probability independent of n. Lower q values for a tree represent higher quality trees. I will use q values to compare trees with very different sizes and costs.

Inverting the formula for q gives us the SAM cost in terms of q: $C_{SAM} = cn^{1+\log_F q}$. It is important to understand that q is a *quality* not a *cost*. C_{SAM} is a cost and will relate directly to the amount of work performed when traversing the acceleration structure. On the other hand, q is a quality metric that describes how "good" a tree is, independent of its size. Tree quality has a valid range of $0 < q \leq c^{\frac{-1}{\log_F n}} \leq 1$ due to the fact that the SAM cost function is bounded by $0 < C_{SAM} \leq n$. I will define a tree as "good" if the quality is $q : q \leq Q$ for a provided quality threshold Q. It should be noted that this notion of "good" does not require trees to have low fan-out.

5.5.8 Quality Analysis of the Algorithm

Given a definition for a "good" acceleration structure, I will now show that the build from hierarchy algorithm will produce "good" acceleration structures from appropriate input structures and assumptions about the scene.

A local strategy for achieving linear-time build using constant-size candidate sets at each step is not guaranteed to produce a high-quality acceleration structure. We must rely on the properties of an input hierarchy to make guarantees about the resulting structure. Given a scene, we would like to show that the build from hierarchy algorithm produces an acceleration structure of comparable quality to a structure produced by the SAH. Specifically, for an input quality threshold $Q : Q \leq 1$, we would like to show that if the SAH would build a tree of quality $q : q \leq Q$, then my build from hierarchy algorithm will also produce a tree of quality $q' : q' \leq Q$.

To prove this property, I will rely on the following assumptions about the input hierarchy and scene geometry:

1. The input hierarchy must have SAM quality of $q'' : q'' \leq Q$. This assumption asserts that the input hierarchy is a "rough sort" of the geometry. Without this assumption, it is impossible to construct a high-quality tree in linear time. It should be noted that scene graphs commonly have this property in practice, as do acceleration structures constructed with the SAH for previous frames.

2. A $q'' : q'' \leq Q$ quality acceleration structure exists for every $x : x <$

$a + F_I$ collection of candidate objects for each node in the hierarchy and the SAH will find one such structure. This assumption asserts that all $x : x < a + F_I$ size collections of candidates acquired by refining the scene-graph BVH are well enough behaved that some high-quality structure can be constructed over them. This assumption prevents the occurrence of arbitrarily bad geometry and holds for scenes in practice.

Observation 1: If the SAH will produce a binary tree of quality q for some $q \leq Q$ then the build from hierarchy algorithm will also produce a tree of quality $q' : q' \leq Q$ given assumptions 1 and 2 above.

Proof: The proof will proceed in two parts. First I show that for all $x : x < a + F_I$ collections of refined geometry the algorithm produces a high-quality acceleration structure. We assumed that a full SAH build over each x-size collection will produce a high-quality structure. However, the algorithm does not build each x-size tree independently in one step. It may use additional candidates when choosing deeper splits in the x-size tree. It is obvious however that increasing the number of candidates when choosing a split plane can only decrease the SAM cost (by greed). Therefore the build from hierarchy algorithm produces a high-quality acceleration structure for all x-size subsets of the output structure.

Second, given that the algorithm produces a high-quality tree over all x-size subsections of the geometry, each x-size section has quality $q : q \leq Q$ for some q and the given bound Q. Therefore, in the binary output tree,

every x-size leaf structure has cost $2x^{1+\log_2 q}$. The cost of the second tier x-size structures is:

$$2x^{1+\log_2 q} x^{1+\log_2 q'}, q' \leq Q \tag{5.4}$$

$$= 2x^{1+\log_2 q+1+\log_2 q'} \tag{5.5}$$

$$= 2x^{2+\log_2(qq')} \tag{5.6}$$

$$= 2x^{2+2\log_2 \sqrt{qq'}} \tag{5.7}$$

$$= 2(x^2)^{1+\log_2 \sqrt{qq'}} \tag{5.8}$$

$$\tag{5.9}$$

Let $q'' = \sqrt{qq'} \leq Q$ (geometric mean). Therefore the algorithm produces high-quality trees for x^2 size subtrees. By induction to a height of $\log_x n$, the algorithm produces high-quality n-size output trees.

5.6 Concluding Remarks

In this chapter I have discussed specialized acceleration structures and high-performance build algorithms. Specialized acceleration structures are structures that make specific assumptions in order to provide some benefit (usually performance-related) but give up some amount of generality. After introducing specialized acceleration structures I show how the perspective transform can be used to produce significantly faster structures for eye and shadow rays. I show that the additional overhead of producing these acceleration structures is outweighed by the improvement in performance they provide. Results are presented in the chapter titled Implementation and Results.

In addition to discussing specialized acceleration structures I describe two build algorithms for kd-trees. Fast build algorithms are required to build specialized acceleration structures quickly enough to make them useful. Additional motivation for fast build algorithms is to allow dynamic scenes in ray tracing. In fact, this was my initial motivation. If a scene changes between frames, acceleration structures must also change to reflect these changes. Therefore, supporting dynamic scenes forces acceleration structure build to be an online process. The algorithms I presented in this chapter provided an overall reduction in kd-tree build time of one to two orders of magnitude, making interactive kd-tree build a reality. Online build was voted to be the most important problem in real-time ray tracing at the Siggraph'05 real-time ray tracing course. Due to work by me and others it is no longer a large concern.

Chapter 6

Implementation and Results

Having spent the majority of this document describing algorithms, I now turn to a discussion of system design and performance results associated with those algorithms. This section is dedicated to experimental validation of the presented algorithms. Results presented in this section were primarily gathered from three different systems.

The Razor [19] system was built in collaboration with Gordon Stoll, Bill Mark, Peter Djeu, Rui Wong and Ikrima Elhassan. My primary contribution to this system was the kd-tree build, including the fast scan and build from hierarchy algorithms. Results I present for those two algorithms were gathered in this system.

Results relating to the perspective grid acceleration structure were obtained using a system I developed independently from other projects. This system was designed to have extremely high performance with the primary intent of demonstrating the performance potential of a ray tracing system for primary visibility and hard shadows.

Comparisons and results relating to the perspective surface area metric were gathered from a third system. This system was designed primarily to be

simple, with performance as a secondary factor. It was carefully designed to allow identical build, traversal and intersection code to be used to generate images in both world space and perspective space.

The same system used to gather perspective surface area metric results was used to gather results for the mailbox corrected surface area metric. Additionally, I used PBRT [54] to further experimentally validate the mailboxing correction.

6.1 Correction to the SAM for Mailboxing

Analysis using the Original SAM (referred to as the OSAM) can bring to light fundamental differences between kd-trees and axis aligned BVHs. In particular, BVHs have substantially nicer behavior with respect to duplication. The crux of the difference is that object duplication in spatial data structures is discrete. If an object is referenced multiple times by a partitioning acceleration structure, the duplication factor is an integer. On the other hand, when space is duplicated in an aggregation acceleration structure, the volume of duplication is real-valued. The resulting fact is that an order ϵ overlap in a kd-tree causes an entire object to be duplicated instead of an order ϵ volume of space. The mailboxing optimization partially addresses this problem in kd-trees. However, without a modification to the SAM, this optimization isn't as effective as it could be.

6.1.1 BVHs, KD-Trees and Mailboxing

In this subsection, I present an example that highlights the previously described difference in behavior between BVHs and kd-trees. I will examine how mailboxing changes the behavior of kd-trees when using the modified heuristic presented in the chapter about cost metrics. For this section we will set C_s (the cost of traversing a node) to zero. Under this assumption, the SAM cost is equivalent to the expected number of intersection tests performed by a ray during traversal. Thus, this section describes how the modified metric behaves when considering only the expected number of intersection tests. It should also be noted that C_s is very small in a kd-tree (when compared to ray-triangle intersection or to BVH node traversal), especially when ray packets are used to amortize loading and branching penalties.

This section is divided into three different cases. In each case, the kd-tree will exhibit different behavior. The cases are ordered to provide intuition as to how my correction to the SAM for mailboxing changes the behavior of the SAH. These examples are in two dimensions (making the figures clearer), but the generalization to three dimensions should be obvious. Since the examples are in two dimensions, the SAM will use perimeters as surface areas. Also, because surface areas are only used in ratios (to compute probabilities), the examples use $\frac{1}{2}$ of the perimeter to make the math simpler (the factors of two cancel in the ratio).

6.1.1.1 Simple Case

The first case is the simple case of two axis-aligned boxes that are abutted. See Figure 6.1. A BVH constructed over these boxes is simply the boxes themselves and a parent box. The cost of such a structure using the traditional cost model is:

$$\begin{aligned} cost &= C_L P_L + C_R P_R && (6.1) \\ &= 1\frac{1+1}{2+1} + 1\frac{1+1}{2+1} && (6.2) \\ &= \frac{4}{3} && (6.3) \end{aligned}$$

A kd-tree will have one splitting plane and identical cost to the BVH (probabilities and costs are identical).

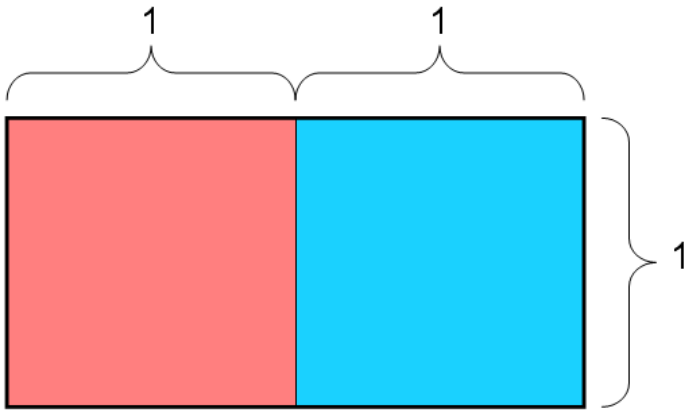

Figure 6.1: A simple case involving two objects (boxes) abutted.

6.1.1.2 Perturbed Case

An interesting thing happens when we slightly alter the boxes by skewing the divider between them. See Figure 6.2. In the case of the BVH there is a slight overlap between the two child bounding volumes and a small amount of space becomes duplicated. The cost of this structure using the traditional cost model is:

$$\begin{aligned} cost &= C_L P_L + C_R P_R && (6.4)\\ &= 1\frac{1+(1+\Delta)}{2+1} + 1\frac{1+(1+\Delta)}{2+1} && (6.5)\\ &= \frac{4+2\Delta}{3} && (6.6) \end{aligned}$$

This cost is very similar to the cost for the non-perturbed case. The difference between the two costs is $O(\Delta)$ as we would expect. In the case of a kd-tree (see Figure 6.3), something very different happens. An object becomes duplicated across the partition and the cost of the structure jumps to:

$$\begin{aligned} cost &= C_L P_L + C_R P_R && (6.7)\\ &= 1\frac{1+(1-\Delta)}{2+1} + 2\frac{1+(1+\Delta)}{2+1} && (6.8)\\ &= \frac{6+\Delta}{3} && (6.9) \end{aligned}$$

This cost is actually *greater* than the cost of not splitting at all. (The cost of not splitting is 2). The OSAM will not allow this split to be performed. A small perturbation of the original example produced a small perturbation in the cost of the BVH but a large change in the cost of the kd-tree. Since

volume is a real quantity, we can overlap a very small amount of it, thus small changes in our input may result in small changes in our output. However, the quantity of overlapped objects is a discrete quantity and cannot change by a "small" amount. The minimum overlap in the number of objects is one. In this case one, is a large fraction of the total objects present. Due to this effect, rays traversing a BVH perform fewer (or the same number of) intersection tests than do rays traversing a kd-tree over dense geometry.

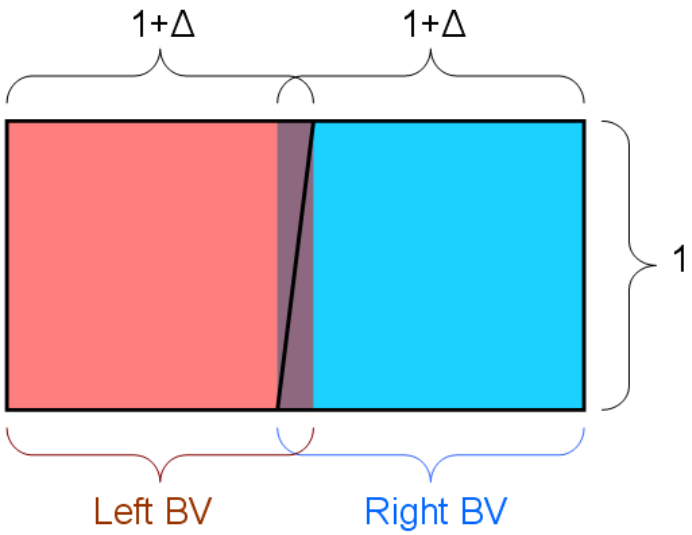

Figure 6.2: A perturbed case involving two objects (trapezoids) abutted.

6.1.1.3 Perturbed Case with Mailboxing

In the third case we revisit the perturbed case but with the addition of mailboxing and my modified surface area metric. Using these changes, the

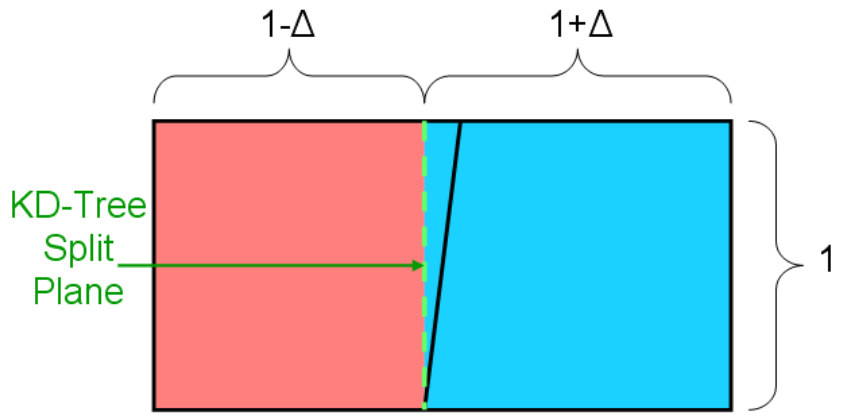

Figure 6.3: A perturbed case involving two objects (trapezoids) abutted with one kd-tree split.

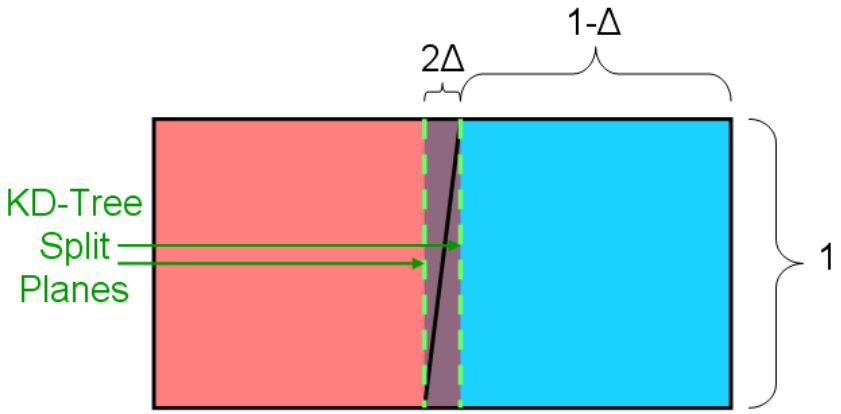

Figure 6.4: A perturbed case involving two objects (trapezoids) abutted with two kd-tree splits.

cost of the kd-tree split is:

$$\begin{aligned} cost &= C_L P_L + C_R P_R - C_{L\wedge R} P_{L\wedge R} && (6.10)\\ &= 1\frac{1+(1-\Delta)}{1+2} + 2\frac{1+(1+\Delta)}{1+2} - 1\frac{1+0}{1+2} && (6.11)\\ &= \frac{5+\Delta}{3} && (6.12) \end{aligned}$$

For values of Δ less than one, the cost is in fact less than the cost of not splitting the node. The modified SAH will make this split, even though the OSAH would not. The cost is still higher than that of the BVH, *but* the kd-tree has the opportunity to make another split. If this second split is allowed, the cost of the right child is reduced to (see Figure 6.4):

$$\begin{aligned} C_R &= C_{L'} P_{L'} + C_{R'} P_{R'} - C_{L'\wedge R'} P_{L'\wedge R'} && (6.13)\\ &= 2\frac{1+2\Delta}{1+(1+\Delta)} + 1\frac{1+(1-\Delta)}{1+(1+\Delta)} - 1\frac{1+0}{1+(1+\Delta)} && (6.14)\\ &= \frac{3+3\Delta}{1+(1+\Delta)} && (6.15) \end{aligned}$$

For small Δ, the cost of splitting the right child is also less than the cost of not splitting it. We may substitute our new value of C_R into equation (6.10) for the parent box. By canceling terms we get:

$$\begin{aligned} cost &= C_L P_L + C_R P_R - C_{L\wedge R} P_{L\wedge R} && (6.16)\\ &= 1\frac{1+(1-\Delta)}{1+2} + \frac{3+3\Delta}{1+2} - 1\frac{1+0}{1+2} && (6.17)\\ &= \frac{4+2\Delta}{3} && (6.18) \end{aligned}$$

Using mailboxing and our new surface area metric, the kd-tree has achieved the same cost as the BVH. A ray is expected to perform the same

number of intersection tests in both structures. By using the modified SAM, we construct a higher quality kd-tree. Since the modified SAM allows additional splits that the OSAM does not, it provides additional opportunities to use the mailboxing optimization, thus making mailboxing more effective. This example demonstrates that the corrected metric can make a difference in the number of expected intersection tests performed during kd-tree traversal by providing new opportunities for partitioning.

6.1.2 Verification Results

I implemented this correction in PBRT [54] and in an interactive ray tracer. Over a variety of scenes the modified metric reduced the number of ray-primitive intersections by approximately 30%. The exception to this improvement is the Stanford bunny which only improved by approximately 20%. The Stanford bunny contains uniformly finely tessellated triangles. These triangles do not span many cells, thus reducing the need for and effectiveness of mailboxing. See Table 6.1. Additionally, the modification improves end-to-end runtime by a few percent. Importantly, it increases the number of traversal steps by only a small amount, providing a favorable tradeoff between traversal steps and intersection tests in the results. These results demonstrate that the correction provides a practical improvement, providing noticeable benefits to both the number of intersections performed as well as overall runtime, while only increasing the number of traversal steps by a small amount.

Scene	Intersection Tests		
	Original	Modified	Change
Plants	1,263.7 M	872.9 M	-31%
Sibenik	257.3 M	188.4 M	-27%
TT	76.6 M	53.7 M	-30%
Bunny	14.76 M	11.66 M	-21%
Fiary Forest	29.43 M	21.60 M	-27%
Scene	Time		
	Original	Modified	Change
Plants	716.8 s	682.5 s	-4.8%
Sibenik	219.6 s	211.9 s	-3.5%
TT	102.7 s	101.4 s	-1.3%
Bunny	1.768 s	1.684 s	-4.8%
Fiary Forest	3.359 s	3.006 s	-10.2%
Scene	Ray Traversal Steps		
	Original	Modified	Change
Plants	14.18 B	14.70 B	+3.67%
Sibenik	1.934 B	2.006 B	+3.72%
TT	856 M	922 M	+7.71%
Bunny	43.63 M	46.69 M	+7.01%
Fiary Forest	96.10 M	99.54 M	+3.58%

Table 6.1: Improvements due to the modified SAM. Results for Plant, Sibenik and TT use PBRT in its default configuration on a 2.66 Ghz Core2. Bunny and Fairy Forest results use an interactive kd-tree ray tracer and scene/camera files from [35] on a 2.2Ghz Core2 (Merome). (Model source/details are covered in the second appendix.) Results were run at 1920x1200 in single ray mode using inverse mailboxing with a 16-entry cache. Columns show improvements in number of intersections and runtime as well as the increase in number of traversal steps made by rays using the modified structure.

6.2 The Perspective Grid

This section describes system implementation details for the perspective grid ray tracer I developed. My system is used to obtain the performance results also presented in this section.

6.2.1 System Design

As in a standard Whitted ray tracer, the overall flow of control in the system is driven by the tracing of eye rays, which in turn trigger the tracing of shadow rays. Throughout this section I will discuss how to use the perspective grid to achieve high performance for different sets of rays using my system as an example.

6.2.1.1 Eye Rays

Eye rays are exceptionally well behaved: they share a common origin, their directions are evenly distributed and can be computed from a formula rather than stored, and these rays can easily be grouped in any desired way. Thus, the acceleration structure and traversal algorithm may be highly optimized for these conditions. With these optimizations, the algorithm for eye rays can be considered to be either a degenerate (no splits in z) version of the perspective grid technique or a modified tiled z-buffer renderer of the sort-middle variety [53] with additional capabilities such as the ability to render non-polygonal geometry.

The implementation of the acceleration structure for eye rays is a per-

spective grid with no splits in the z dimension. In the current implementation the grid uses a resolution such that each cell is approximately 100x100 pixels. This grid resolution ensures that the color and depth values for ray hit points associated with a cell fit into the processor's L2 cache. For high depth-complexity scenes, the third dimension of the grid could be restored. The system does not preprocess scene objects for faster intersection. It should be noted that by keeping each cell in the processor's L2 cache, the perspective grid system avoids reading/writing to main memory during the inner rendering loop. This optimization prevents the limited bandwidth to main memory on current generation CPUs from being a bottleneck.

After building the acceleration structure, the system processes one grid cell at a time, along with all of the geometry and eye rays that intersect that cell. Within each cell, intersection tests are performed as they would be in a z-buffer rasterizer and in particular are performed in unsorted object order rather than ray order. For each object, the system finds all rays that intersect the perspective-aligned bounding box of that object and intersects the rays with that object. This intersection is computationally efficient because primary rays may be found/computed via a formula. Rays are processed in packets of 4 using the x86 4-wide SIMD instructions. A per tile distance buffer is checked and conditionally updated for each intersection test. A floating-point color buffer is also updated with a color computed with a simple shading model (dot product of normal with light vector). These intersection and shading algorithms are very similar to the ones used by a tiled z-buffer renderer.

After shadows/shading, simple tone mapping based on min/max intensity from the entire previous frame is performed for all rays in a grid cell, converting 32-bit float per component color down to 8-bit per component color before being written to system memory. As the Results section will show, the performance of this algorithm is much faster than traditional ray tracers and comparable to that of software z-buffer renderers.

6.2.1.2 Hard Shadow Rays

After processing the geometry in a grid cell to find all intersection points, the system casts shadow rays for all hit points found in that cell. However, if the intersection point is on a back-facing polygon (with respect to the light), it is assumed to be in shadow without tracing the shadow ray.

Hard shadow rays are almost as well behaved as light rays. However, ray directions must be stored explicitly since they are not regularly spaced. Also, intersection testing is done in ray order since shadow rays are generated on the fly from eye rays. Thus, the system's processing of hard shadow rays is much more like traditional ray tracing than the processing of eye rays.

The system uses one perspective grid per light. These grids also have no splits in the z dimension. Thus, as with eye rays, each hard shadow ray traverses exactly one grid cell. In the current implementation, the hard shadow perspective grid is 200x200 cells, much finer than that used for eye rays. The perspective grid acceleration structure for each light is built at the start of the frame. Only back-facing objects (with respect to the light) are transformed

and added to the grid. The system assumes that models are closed.

There are several advantages to culling the front-facing triangles [73, 77]. At each grid cell, the system stores the distance between the light and the closest geometric primitive in the cell. Using the distance to the closest back-facing triangle is more aggressive than the distance to the closest front-facing triangle (as would be the case if we were back-face culling). This form of hierarchical culling skips all intersection tests for up to 90% of the non-shadowed rays. Using front-face culling also has the advantage that rays launched from the surface of a front face cannot self-intersect, eliminating shadow acne [78].

When rendering a scene with many lights and lots of geometry, it is clear that the use of multiple acceleration structures will consume more memory than a single acceleration structure would. In cases where memory consumption is a problem, it could be addressed in a variety of ways, including building the acceleration structures on demand or by only maintaining one hard shadow acceleration structure at a time. This second approach requires either storing primary hit locations before casting any shadow rays or re-casting primary visibility rays for each light (which is particularly inexpensive using the perspective grid).

As the results section will show, the performance of this algorithm is much faster than traditional ray tracers. Traditional z-buffer renderers cannot support this visibility query at all. Shadow mapping [76] can be used to approximate the result, but causes significant artifacts. The Irregular z-buffer [4, 43] can support this visibility query with high performance and with-

out artifacts, but it uses an object-order system organization which integrates poorly with other ray tracing queries.

6.2.1.3 Soft Shadow Rays

Soft shadow rays (Figure 6.5) require a full 3D traversal of the perspective grid acceleration structure and thus provide the best example of the fact that a perspective grid is a true 3D acceleration structure, capable of supporting traversal by any ray. Here I provide details about the specific implementation details I use for the perspective grid when tracing soft shadow rays. Recall that the perspective grid is a truly 3D uniform grid acceleration structure.

The acceleration structure is a 3D perspective grid (200 x 200 x 4). There are fewer splits in the third dimension, as suggested by the theoretical analysis provided in the Surface Area Heuristic section. As before, each light has its own perspective grid acceleration structure. The acceleration structure is built at the start of the frame, ignoring fully front-facing triangles (i.e. front facing from all points on the light).

As with hard shadow rays, the soft shadow rays are processed in batches of approximately 100x100 driven by the eye rays, but there are now 8 shadow rays per eye hit point. Rays are traversed in 4-wide packets for SIMD efficiency, with the packets consisting of four shadow rays from a single eye ray hit point. This choice of packet construction guarantees that shadow packets are always as coherent as the area light allows. The traverser is a slice-based ray packet

traverser for the grid acceleration structure, following the grid packet technique described by Wald [70].

As the results section will show, traversing the perspective grid is substantially cheaper than traversing a standard 3D grid. As expected, performance is best for small lights and degrades as the light becomes larger because the shadow rays are less well aligned to the primary projective axis. An image demonstrating soft shadows rendered using the perspective grid is provided in Figure 6.5.

6.2.2 Results

This section presents results gathered from the system I have built to implement the perspective grid algorithms and compares these results to alternative approaches. First, I will present the performance of the system on various scenes for eye rays, hard shadows, and soft shadows. Second, I will compare the system's performance to that of other interactive ray tracing systems and software z-buffer renderers. Third, I present operation counts showing that soft shadow rays traversed through a perspective grid require fewer traversal steps than the same rays traversed through a world space grid.

Figure 6.5: Soft shadows rendered by the system (Courtyard scene).

End to End Runtime Results for My System							
Scene	Polys	Primary			Hard Shadows		
		FPS	build	Mray/s	FPS	build	Mray/s
Courtyard	31k	30	5%	68	8	10%	36
FairyForest020	174k	17	17%	39	6	33%	27
Bunny-69k	69k	44	18%	100	11	27%	50
Bunny-16k	16k	81	7%	185	24	21%	109
Bunny-4k	4k	120	3%	274	31	13%	141
Bunny-1k	1k	154	1%	351	37	10%	169
Dragon-Bunny	252k	17	25%	39	4	50%	18
Conference	282k	16	33%	36	5	45%	23
ERW6	1k	56	1%	128	15	3%	68
Subset of the 8-Cores in Parallel Performance Table (Secondary Result)							
Courtyard	31k	150	27%	342	34	41%	155
FairyForest020	174k	55	56%	125	14	78%	64
Bunny-16k	16k	339	31%	773	71	62%	324

Table 6.2: Performance of the perspective grid rendering system for various scenes and for various quality settings, all at 1920x1200 resolution. The results include the time for per-frame build of the acceleration structure. The top nine rows use one core of a 2.66 GHz Xeon X5355, 1333MHz FSB. Hard shadows use one eye ray and one hard shadow ray per pixel. Hard shadows use a 200x200x1 perspective grid, except FairyForest (800x800x1) and Conference (600x600x1). The bottom three rows show parallel performance on eight of the same cores. All phases of the system except acceleration-structure build parallelize well, but since build is not yet parallelized it becomes the bottleneck in the parallel scenario.

End to End Runtime Results for My System				
Scene	Polys	Soft Shadows		
		FPS	build	Mray/s
Courtyard	31k	0.26	0%	5.3
FairyForest020	174k	0.11	1%	2.3
Bunny-69k	69k	0.41	1%	8.4
Bunny-16k	16k	0.57	0%	11.7
Bunny-4k	4k	0.65	0%	13.3
Bunny-1k	1k	0.72	0%	14.8
Dragon-Bunny	252k	0.30	4%	6.2
Conference	282k	0.19	1%	3.9
ERW6	1k	0.19	0%	3.9
Subset of the 8-Cores in Parallel Performance Table (Secondary Result)				
Courtyard	31k	2.0	2%	41
FairyForest020	174k	0.51	4%	15
Bunny-16k	16k	4.2	3%	86

Table 6.3: This table uses the same performance setup as the previous table. Soft shadows use one eye ray and eight Monte Carlo soft shadow rays per pixel and a 200x200x4 perspective grid.

I demonstrate performance many times faster than other dynamic ray tracing systems across a range of scenes. For several scenes I also demonstrate performance of greater than 100 million rays per second for primary visibility on a single core. Additionally, I achieve real-time performance (over 30fps) at 1920x1200 for hard shadows on a game-like scene (courtyard) using eight cores. Finally, the eye ray performance approximately matches that of modern software z-buffer renderers. To the best of my knowledge, no other real-time ray tracing system has these capabilities.

6.2.2.1 Overall Performance

Tables 6.2 and 6.3 shows measured system performance for a variety of models on a single CPU core. The perspective grid system is focused on measuring visibility performance and so I exclude expensive local shading operations that would make the results more difficult to interpret. In particular, I do not implement texture mapping because modern CPUs lack the memory bandwidth and specialized hardware needed for high-performance texture mapping. I do implement per-pixel diffuse shading of interpolated artificial colors to demonstrate that the technique supports interpolated vertex parameters efficiently. The system can interpolate normals similarly, but this capability is disabled by default because many of the scenes lack correct per vertex normals. I report performance for regular eye rays (Table 6.2), hard shadows with regular eye rays (Table 6.2), and soft shadows with regular eye rays (Table 6.3). Several conclusions can be drawn from these results.

First, hard shadow rays are not as fast as eye rays by the metric of ray-segments per second but still perform very well compared to other ray tracing systems. See Table 6.4 for a comparison between dynamic ray tracing systems.

Second, soft shadow rays perform substantially slower than hard shadow rays by the metric of ray segments per sec. There are a couple of reasons for the performance degradation: Most soft shadow rays are not perfectly parallel to the z axis, and so they must make traversal steps in x and y as well as z. The mere fact that the traversal algorithm supports stepping in x, y, and

z makes it significantly slower than the special case algorithm used for hard shadow rays, even if it is used to trace rays parallel to the z axis which do not step. Also, as the size of the light increases, the performance of the perspective grid for soft shadows reduces very rapidly. This is primarily due to the large number of "horizontal" steps through the high resolution grid. Additionally, the perspective grid suffers from the ailment of all uniform acceleration structures known as the "teapot in a stadium" problem. The PSAH presented in the cost metrics chapter of this dissertation addresses soft shadows using adaptive perspective space acceleration structures [35].

Table 6.2 also reports the fraction of the frame time that was spent on building the acceleration structure(s), and these results show some interesting trends. First, the acceleration structure build is somewhat more costly for hard shadows than for primary rays because as discussed earlier the acceleration structure used for shadow rays is more complex than that for primary rays. Second, for models of 100k+ polygons, build cost is a significant fraction (0.10-0.50) of the total frame time for eye rays and hard shadows, but not for soft shadows. Since grid acceleration structures are typically less efficient for traversal than adaptive data structures such as kd-trees, these results suggest that soft shadows would benefit from the additional computation required to build an adaptive acceleration structure such as a perspective kd-tree or perspective bounding-volume hierarchy. Adaptive perspective space build algorithms are explored in the next section of this chapter and in my IEEE RT08 paper[35].

To achieve real-time frame rates on the models used in typical interactive applications, the techniques used in this paper would need to be parallelized for multi-core architectures. I have already parallelized the traversal algorithm, and as expected we observe good scaling (around 90% of linear with 8 cores). Parallelizing the construction of the perspective grid acceleration structures is still future work, but I believe that a combination of ideas from traditional z-buffer parallelization [53], regular-grid parallelization [41] and on-demand parallel build of acceleration structures from hierarchy [37] can be successfully applied to this problem. I provide results for some scenes using eight cores.

6.2.3 Comparison to other ray tracing systems

In this subsection I compare the performance of the perspective grid system, against the performance of other recent high-performance ray tracing systems for eye rays and hard shadow rays. Table 6.4 compares the performance of the system against other systems that support dynamic and semi-dynamic scenes. Table 6.5 provides a similar comparison for static scenes (i.e. excluding acceleration structure build time for comparison systems but not mine). I make a good faith effort to make fair comparisons, but for a variety of reasons – especially incomplete data in previous publications – it is difficult to make such comparisons completely precise or exhaustive. In both of these tables, I adjust previously published results measured on Pentium 4 (3.2 GHz) systems and Opteron (2.6 GHz) systems upward by a conservative

factor of 1.5x to estimate their equivalent performance on the 2.66 GHz Core2 system that I use. This adjustment allows each system to be evaluated on the platform for which it was optimized. The viewpoints used for gathering my results have been visually matched to be as close as possible to the ones used in the publications I compare to. I also use the same resolution as the previous results, 1024x1024.

I first compare my system to Wald's grid system which uses an ordinary grid acceleration structure [70] and thus supports arbitrary dynamic scenes like my system. My system is faster than the grid system by a factor of 2.2x for eye+hard shadow rays on the one model (Fairy Forest) for which I can compare. For eye rays, my system is even faster, by over 4.6x in all cases for which data is available. More precisely, when comparing my system *including* build time against the original grid paper *excluding* build time, my system is 4.6x to 5.0x faster. With build time excluded for both systems, my system is 5.1x to 7.9x faster.

I also compare my system to Wald's bounding volume hierarchy system [67], which can be considered to be semi-dynamic because the topology of its optimized bounding volume hierarchy is precomputed in a slow preprocessing step. Only the bounds in the hierarchy are updated each frame. My system is always faster than the BVH system for eye rays, with conservative speedup ranging from 1.37x to 2.83x. My system is faster than the BVH system in most cases for eye+shadow rays, with speedup ranging from 0.94x (i.e slight slowdown) to 2.36x. The largest speedups are seen with the Fairy Forest

model, which is the only truly animated model and thus is the most reasonable point of comparison.

Finally, in Table 6.5 I compare my system to MLRTA [58], a high performance ray tracer for static scenes which uses a highly optimized pre-built acceleration structure. With my system rebuilding its lightweight acceleration structure every frame, it matches or outperforms MLRTA. Therefore, my perspective grid system demonstrates that dynamic ray tracing can be as fast as the fastest current static ray tracing systems.

Unfortunately, I cannot compare my system to any other high performance rendering systems on the Courtyard scene which I believe to be the best proxy for modern game scenes. My system performs especially well on this scene for a variety of reasons, including the scene's relatively uniform polygon size.

Overall, the perspective grid acceleration structure provides z-buffer like performance with some additional ray tracing flexibility. The results demonstrate that this structure enables performance for eye and shadow rays that is significantly higher than other fully dynamic ray tracing acceleration structures. It also demonstrates competitive soft shadow performance despite suffering from the problems of uniform structures. These results demonstrate that the perspective grid is a very appropriate acceleration structure for near point origin visibility queries.

		Eye Rays			Eye + Hard Shadow Rays		
Scene	Polys	Us	grid	BVH	Us	grid	BVH
ERW6	1K	106	–	∼64	36	–	∼23
Fairy	174K	26	–	∼9.2	7.6	∼3.4	∼3.2
Conf	282K	22	–	∼16	6.8	–	∼7.2

Table 6.4: Comparison of dynamic scene performance, all in frames/sec at 1024x1024 resolution. My system is running on one core of a 2.66 GHz Xeon 5355; Other results are taken from recent publications with different hardware, but adjusted in this table to estimate performance on a single 2.66 GHz Xeon 5355. Notes: (1) Adjustment for processor differences is an estimate; see text. (2) The BVH algorithm is restricted to certain types of dynamic scenes because it computes its acceleration structure topology off-line (taking an adjusted 2.16 sec for Fairy) and only updates the bounds each frame. (3) The BVH render times include basic texture mapping; others do not.

6.2.4 Comparison to z-buffer systems

When my system is only tracing eye rays, its functionality and algorithms for visibility are similar to those of a tiled z-buffer renderer. Thus, it is appropriate to compare my system's performance to an optimized software z-buffer renderer. In Table 6.6, I compare to Pixomatic 2.0 [3], which is sold commercially by RAD software for use as a fallback renderer in PCs lacking fast graphics hardware. RAD software reports performance results for their system on a 3.3 GHz Pentium-4 [56], and I adjust these results upward by a factor of 1.5x to estimate performance on the 2.66 GHz Xeon X5355.

I measure performance using the 69.5K bunny model under conditions that are as similar as possible to those used by RAD given the different natures of the two systems. It is important to note that Pixomatic is configured to

		Eye Rays			
Scene	Polys	Us	MLRTA	grid	BVH
ERW6	1K	106	~ 76	~ 21	~ 49
Fairy	174K	26	–	–	~ 12
Conf	282K	22	~ 23	~ 4.8	~ 14

Table 6.5: Comparison of static scene performance, all in frames/sec at 1024x1024 resolution. Results for my system include the time for acceleration structure build (since it is view dependent), while results for other systems exclude build time. My system is running on one core of a 2.66 GHz Xeon 5355. Other results are taken from recent publications with different hardware, but adjusted to estimate performance on the 2.66 GHz Xeon 5355.

	My System	Pixomatic (x1.5)
Triangle Rate	8.32	∼7.92
Transform & Project Rate	32.1	∼35.0

Table 6.6: Comparison of my system's performance for eye rays vs. published results for Pixomatic, a high performance software z-buffer renderer. Notes: (1) Pixomatic is performing texture mapping but my system is not; see text for details. (2) Pixomatic results are adjusted upward to account for hardware differences.

perform texture mapping while my system is not. However the effects of this difference should be small because the triangle rate in both systems is measured with only 5% of the window covered, and the transform and project rate is measured with the model off-screen. Thus, although these results must be interpreted with caution, I believe they do show that the performance of my system for eye rays is comparable to that of a high performance software z-buffer renderer.

	Regular 3D Grid	Perspective 3D Grid
Grid Size	54x54x54	200x200x4
Traversal Steps/Ray	52.0	12.7

Table 6.7: When tracing soft shadow rays, the perspective grid requires fewer grid traversal steps than a regular 3D grid. These results are measured using my system with the Courtyard scene at 1920x1200 resolution.

6.2.5 Perspective Grid vs. Regular Grid

Although execution time is the ultimate metric for most real-time rendering algorithms, more specific measurements can provide deeper algorithmic insights. To assess the effectiveness of the perspective grid acceleration structure as compared to an ordinary grid, in Table 6.7 I compare the number of traversal steps used when rendering soft shadows for the Courtyard scene at 1920x1200. The total number of cells in each grid is essentially the same but the results show that the perspective 3D grid requires less than 1/3 the number of traversal steps required by the regular 3D grid. I have not implemented a high-performance world-space grid traversal algorithm and thus cannot directly compare the performance between my own regular and perspective grids. See my comparisons to Wald's highly optimized grid algorithm in the previous section.

6.3 The Perspective Surface Area Metric

Having discussed the perspective grid and its performance, this section presents and discusses the implementation and results related to the perspec-

tive surface area metric. I implemented a ray tracing system to evaluate the effectiveness of adaptive perspective space acceleration structures. The ray tracer is a simple, SIMD(4)-wide packet ray tracer [72] using a kd-tree as the acceleration structure. The primary goal of the implementation is to determine the effectiveness of building and using adaptive perspective space acceleration structures versus "regular"-space structures. Great care was taken to ensure that identical build code (except for heuristic evaluation), traversal code and intersection code was used in both regular and perspective space for the fairest comparison.

Geometry is transformed into perspective space and clipped during the frame-setup phase, before being handed to the build algorithm for the acceleration structure. This frame setup cost *is* counted in the **Build Time** column in Table 6.10. Rays are transformed into perspective space before being traversed. Shadow rays that use a different perspective space than primary rays are transformed between the perspective spaces in a similar method to that used in z-buffer shadow mapping.

The results in Table 6.10 and Table 6.11 compare the performance of the regular kd-tree versus a perspective kd-tree in two common scenes (bunny69k and fairy-forest) from the viewpoints illustrated in Figure 6.6. Results are provided for eye rays, hard shadow rays, soft shadow rays, and depth-of-field eye rays. Figure 6.7 provides details about the soft shadow and depth-of-field images. The fairy forest model is approximately 6.3 x 1.6 x 6.3 units in size. Depth of field in the fairy forest scene is computed using an aperture "radius"

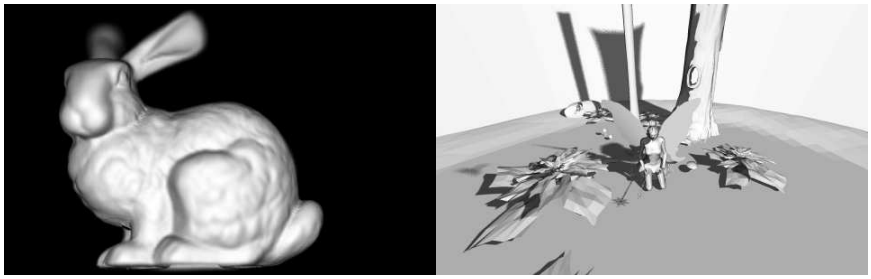

Figure 6.6: The Stanford Bunny rendered with depth of field and Fairy Forest rendered with soft shadows both using a perspective kd-tree. The Fairy Forest scene is approximately 6.3 x 1.6 x 6.3 units in size. The light "radius" for the result is 0.1.

of 0.01 and a focal distance of 0.91. Soft shadows in the fairy forest scene are computed using a light with "radius" 0.1. For the bunny image, the model is approximately 0.15 x 0.15 x 0.12 units in size, the focal depth is 1.98 and the aperture "radius" is 0.1. In all cases an aperture or light of radius x is a $2x$ x $2x$ quad. Back-face culling in the perspective space structure reduces build time by approximately 50% in each perspective case. On average, the perspective space acceleration structures, along with the PSAH reduce render times by 20-40% over the traditional SAH and regular space structures. The results demonstrate that even for such a simple case as eye rays, the improvement in trace performance makes up for the additional build time providing end-to-end runtime performance in perspective space similar to render time (excluding build) in regular space. More noticeably, the perspective space acceleration structures reduce the number of intersection tests by about 40% on average. This result demonstrates that the perspective space acceleration structure is of much higher "quality," i.e. it reduces the number of intersection

tests performed.

If the area light is scaled to be very large, the benefits from the perspective transform shrink accordingly. Particularly, if an object is closer to a light than several times the radius of that light, the transform is less likely to help. The results demonstrate the effectiveness of the transform for common scenarios with a reasonably confined light source at reasonable distances. For scenes with very large light sources I would argue against using ray tracing at all. Ray tracing is an inherently high frequency operation and extremely soft lighting is inherently low frequency result. Ray tracing solutions to low frequency computations tend to be either extremely expensive to compute or have very high levels of error (usually in the form of noise). Many, more appropriate algorithms exist for computing low frequency lighting. Low frequency algorithms are not a focus of this dissertation.

When the number of rays traced is large, perspective space acceleration structures more than compensate for their build cost by improving trace performance. Thus, each area light in a scene should use its own perspective based acceleration structure in order to maximize overall system performance. Memory concerns related to having multiple acceleration structures and large scenes may be addressed by only maintaining one such structure at a time as is described in my IEEE RT08 paper [36] and in this dissertation.

In Table 6.11, I compare the performance of different build heuristics in perspective space. The table demonstrates that the PSAM is a significantly more effective cost-estimation metric than either the traditional SAM

Fairy Forest Scene (Static Counts)		
Acceleration structure	Intersections	kd-tree steps
Eye Rays		
(1) traditional / SAH sort	9.369 M	23.15 M
(2) traditional / SAH scan	9.530 M	24.77 M
(3) perspective / PSAM scan	5.482 M	14.49 M
ratio: (3)/(2)	58%	58%
Depth of Field x16		
(1) traditional / SAH sort	148.0 M	369.6 M
(2) traditional / SAH scan	150.6 M	395.5 M
(3) perspective / PSAM scan	92.60 M	314.7 M
ratio: (3)/(2)	61%	80%
Hard Shadows		
(1) traditional / SAH sort	21.05 M	46.13 M
(2) traditional / SAH scan	21.39 M	49.78 M
(3) perspective / PSAM scan	12.29 M	39.33 M
ratio: (3)/(2)	57%	80%
Soft Shadows x16		
(1) traditional / SAH sort	346.4 M	558.9 M
(2) traditional / SAH scan	347.7 M	601.6 M
(3) perspective / PSAM scan	204.7 M	540.4 M
ratio: (3)/(2)	59%	90%
Bunny69K Scene (Static Counts)		
Acceleration structure	Intersections	kd-tree steps
Eye Rays		
(1) traditional / SAH sort	2.838 M	10.43 M
(2) traditional / SAH scan	2.888 M	11.10 M
(3) perspective / PSAM scan	1.515 M	5.981 M
ratio (3)/(2)	52%	54%
Depth of Field x16		
(1) traditional / SAH sort	81.19 M	255.1 M
(2) traditional / SAH scan	83.15 M	272.4 M
(3) perspective / PSAM scan	46.29 M	141.1 M
ratio: (3)/(2)	56%	52%

Table 6.8: Static performance Results.

Fairy Forest Scene Performance			
Acceleration structure	Build Time	Render Time	Total Time
Eye Rays			
(1) traditional / SAH sort	7.21 s	1.16 s	8.37 s
(2) traditional / SAH scan	0.76 s	1.22 s	1.99 s
(3) perspective / PSAM scan	0.58 s	0.71 s	1.29 s
ratio: (3)/(2)	75%	58%	65%
Depth of Field x16			
(1) traditional / SAH sort	7.24 s	18.5 s	25.8 s
(2) traditional / SAH scan	0.78 s	19.1 s	19.9 s
(3) perspective / PSAM scan	0.58 s	14.5 s	15.1 s
ratio: (3)/(2)	74%	76%	76%
Hard Shadows			
(1) traditional / SAH sort	7.28 s	2.13 s	9.41 s
(2) traditional / SAH scan	0.80 s	2.34 s	3.15 s
(3) perspective / PSAM scan	1.15 s	1.86 s	3.01 s
ratio: (3)/(2)	143%	79%	96%
Soft Shadows x16			
(1) traditional / SAH sort	7.83 s	35.5 s	43.3 s
(2) traditional / SAH scan	0.79 s	34.4 s	35.3 s
(3) perspective / PSAM scan	1.19 s	24.3 s	25.5 s
ratio: (3)/(2)	138%	71%	72%

Table 6.9: First half of dynamic performance results. Table is continued on next page along with a descriptive caption.

Bunny69K Scene Performance			
Acceleration structure	Build Time	Render Time	Total Time
Eye Rays			
(1) traditional / SAH sort	2.17 s	0.492 s	2.62 s
(2) traditional / SAH scan	0.26 s	0.532 s	0.79 s
(3) perspective / PSAM scan	0.15 s	0.272 s	0.43 s
ratio (3)/(2)	59%	51%	54%
Depth of Field x16			
(1) traditional / SAH sort	2.09 s	12.7 s	14.8 s
(2) traditional / SAH scan	0.25 s	13.3 s	13.5 s
(3) perspective / PSAM scan	0.15 s	7.8 s	8.0 s
ratio: (3)/(2)	62%	66%	59%

Table 6.10: (Second half of Table 6.9.) Comparison of performance: perspective-space kd-tree vs. traditional world-space kd-tree. **"traditional / SAH sort"** is a traditional kd-tree built using the SAH with sort-based selection at every step. **"traditional / SAH scan"** is a traditional kd-tree built using the SAH with a scan-based selection at every step [38]. **"perspective / PSAM scan"** is a perspective-space kd-tree built using the perspective-space cost metric with a scan-based selection at every step. All results are at 1920x1200 resolution on a single core of a mobile 2.2 Ghz Intel Core 2 Duo (Merom). **ratio** is the ratio of the perspective-space results to the traditional scan-based results. Build time results include all build times (if more than one structure is used). All acceleration structures specialized to a light or camera use face culling.

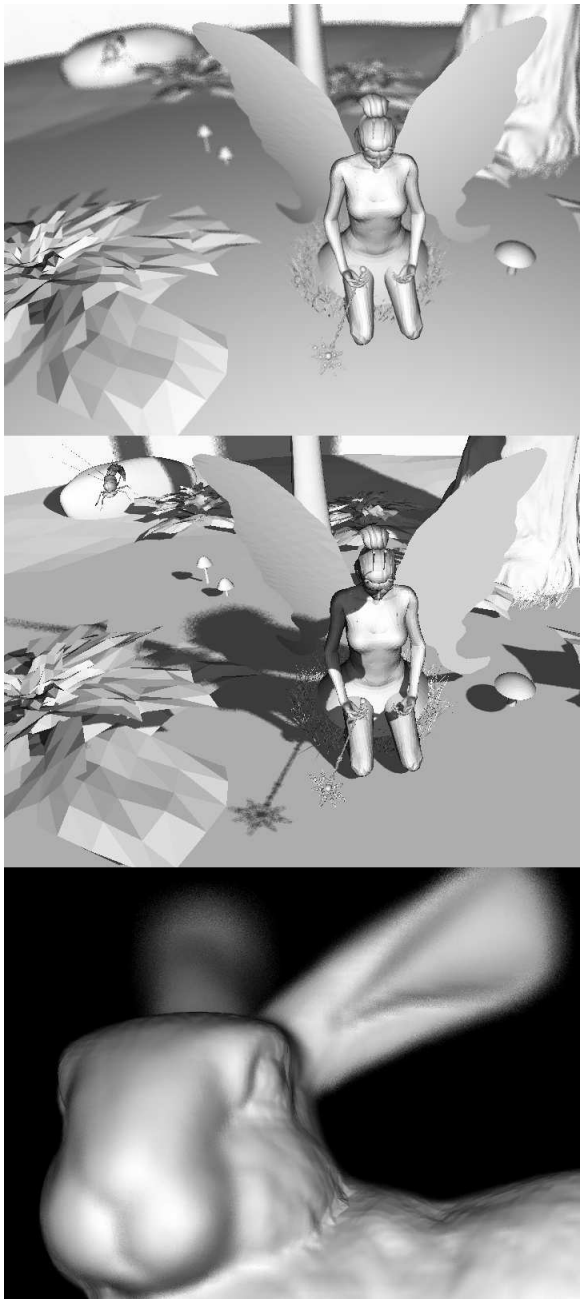

Figure 6.7: Close-ups of depth of field and soft shadow images. Top: Fairy Forest scene with depth-of-field. Middle: Fairy Forest scene with soft shadows. Bottom: Bunny scene with depth-of-field.

Fairy Forest Scene

Build heuristic	Build	Intersections	kd-tree steps	Render	Total
Eye Rays					
Median split	0.209 s	63.85 M	48.76 M	4.16 s	4.38 s
SAH	0.449 s	11.46 M	31.64 M	1.43 s	1.88 s
PSAM	0.581 s	5.482 M	14.49 M	0.71 s	1.29 s
Depth of Field x16					
Median split	0.208 s	999.1 M	794.4 M	65.8 s	66.0 s
SAH	0.472 s	177.2 M	519.5 M	23.3 s	23.8 s
PSAM	0.580 s	92.60 M	314.7 M	14.5 s	15.1 s

Table 6.11: Comparison of different build heuristics for a perspective-space kd-tree. **Median-split** uses the median split of the longest axis. **SAH** uses the traditional SAM, but in perspective space. **PSAM** uses this dissertation's new perspective-space PSAM. All results are at 1920x1200 resolution on a single core of a mobile 2.2 Ghz Intel Core 2 Duo (Merom).

in perspective space or median split of longest axis in perspective space.

6.4 Fast Scan

In this section I discuss the implementation and results for the fast scan approximation to the SAM introduced in the specialization chapter. This discussion contains both technical details and results. I implemented a SIMD-vectorized (SSE2) version of the kd-tree building algorithm as a stand alone package and evaluated it both for build performance and tree quality. I then implemented the same algorithm within the Razor system [19] in order to evaluate the practical effect of the tree quality on render time. The results in Table 6.13 demonstrate that the algorithm's kd-tree build performance is competitive with other interactive acceleration structure builds (circa 2006) and

that the quality of the resulting kd-trees is almost as high as those produced by a sorting method.

Although the Razor system normally uses a lazy kd-tree builder, it was modified to force a full (non-lazy) kd-tree build to make the measurements presented here. Rendering time was measured with the system configured to use large numbers of both primary (4x super-sampling) and secondary (area light sources) rays. Because Razor has some fixed per-ray overhead for multi-resolution ray tracing, a non-multiresolution ray tracer would likely see somewhat larger percentage changes in rendering times than those shown in Table 6.13.

Several of the results in Table 6.13 are outliers deserving further discussion. First, the courtyard scene has an especially low SAH cost (i.e. high tree quality). The primary cause of the low cost is the fact that the scene is largely axis-aligned. Second, the soda hall scene has high SAH cost (i.e. low tree quality) and a fast build time. I believe this behavior is caused by the large wall polygons in the scene which cause the SAH to terminate earlier, producing larger leaf nodes. Finally, Table 6.13 is missing rendering time results for the armadillo and soda hall scenes due to memory consumption issues in Razor. All of the results presented are for builds from an AABB "soup."

The algorithm switches to exact evaluation of the SAH for nodes below a certain size (36 primitives). This cutoff was chosen to lie at the point where the cost of $O(m^2)$ brute-force search has approximately equal cost to the $O(m)$ two-pass adaptive sampling algorithm.

Scene			
	Hand	Bunny	Armadillo
Geometric Information			
Quads	7,510	0	0
Triangles	835	69,451	345,944
Total Polygons	8,345	69,451	345,944
Build Time in Seconds			
All Axes	0.024	0.25	1.41
Hybrid	0.022	0.23	1.14
One Axis	0.012	0.11	0.69
Build speed in polygons/second			
All Axes	348,000	278,000	245,000
Hybrid	379,000	302,000	303,000
One Axis	695,000	631,000	501,000
Tree Quality, Measured as SAH Cost			
All Axes	70.0	76.0	63.8
Hybrid	70.0	76.0	73.3
One Axis	72.0	95.0	84.6
Increase in Render Time Over Sort Based Tree			
All Axes	0.33%	1.63%	-
Hybrid	0.89%	1.63%	-
One Axis	7.90%	7.72%	-

Table 6.12: Fast scan build performance measurements (Part 1), using one core of a 2.4 GHz Intel Core 2 Duo processor ("Conroe"), with 4MB of L2 cache. Models are discussed in detail in the second appendix.

Scene			
	Fairy Forest	Courtyard	Soda Hall
Geometric Information			
Quads	78,201	141,657	0
Triangles	17,715	37,574	1,510,775
Total Polygons	95,916	179,231	1,510,775
Build Time in Seconds			
All Axes	0.30	0.69	5.14
Hybrid	0.27	0.63	4.50
One Axis	0.14	0.26	1.59
Build speed in polygons/second			
All Axes	320,000	260,000	294,000
Hybrid	355,000	284,000	336,000
One Axis	685,000	689,000	950,000
Tree Quality, Measured as SAH Cost			
All Axes	53.6	39.3	452
Hybrid	59.6	39.9	450
One Axis	69.0	43.7	489
Increase in Render Time Over Sort Based Tree			
All Axes	3.19%	3.31%	-
Hybrid	3.77%	3.60%	-
One Axis	4.35%	3.60%	-

Table 6.13: Fast scan build performance measurements (Part 2), using one core of a 2.4 GHz Intel Core 2 Duo processor ("Conroe"), with 4MB of L2 cache. Models are discussed in detail in the second appendix.

6.5 Build from Hierarchy

This section presents the experimental results for the build from hierarchy algorithm. The results were gathered using the Razor system and demonstrate that the build from hierarchy algorithm can greatly improve build performance and that the measured asymptotic behavior fits with the theoretical results. It should be noted that the Razor implementation of acceleration structure build can be slower than other build implementations. The slowness in Razor's build implementation is related to the complexity of the multi-resolution tree and its associated overhead. This overhead does not affect the relative performance improvements provided by build from hierarchy. It also causes builds to be slightly faster for the smaller visible scene even though the lazy part of the build system is off.

For these results I use a "Courtyard 64" 6.9 scene that has a scene graph hierarchy. It does not, however, have a particularly good hierarchy. For example: the fan-out at several leaf nodes is 1568. Any reasonable scene graph should provide hierarchy as good or better than the one I use. I also use the conference scene 6.10 which has a very similar quality hierarchy to the courtyard 64 scene.

Tables 6.14 and 6.15 show that build from hierarchy, lazy build and scan-based SAM evaluation each contribute to improved build performance. In particular, the combination of lazy build and build from hierarchy is much more effective than either in isolation when the number of visible primitives is much smaller than the total number of primitives. For the $v = 9392$ view

			crtyrd near (n=541023) (v=9392)	crtyrd near	crtyrd far (n=541023) (v=89040)	crtyrd far
Lazy	Scan	Hier	build time	trace time	build time	trace time
+	+	+	0.116 s	1.302 s	1.608 s	1.388 s
+	+	-	0.896 s	1.305 s	1.873 s	1.399 s
+	-	+	0.401 s	1.287 s	3.985 s	1.361 s
+	-	-	3.214 s	1.285 s	9.651 s	1.366 s
-	+	+	3.843 s	1.303 s	3.956 s	1.389 s
-	+	-	4.089 s	1.298 s	4.205 s	1.397 s
-	-	+	6.620 s	1.290 s	6.734 s	1.369 s
-	-	-	12.270 s	1.283 s	12.385 s	1.368 s

Table 6.14: Build times for a k-d tree with various fast-build capabilities enabled and disabled. Results are presented for two viewpoints of the Courtyard_64 scene. These results were taken from the Razor system on a single core of an Intel Core 2 Duo 2.667 GHz machine. The next table contains information about a different scene.

there is a 7.7x speedup when using lazy+scan+hierarchy as compared to just lazy+scan. These results experimentally confirm the theoretical result that an $O(v + \log n)$ build is significantly faster than an $O(n)$ build.

In contrast, the difference between $O(n)$ build from hierarchy+scan and the $O(n \log n)$ build from scan is less significant. This smaller difference is due to the slow growth of $\log n$. Thus the most important practical implication of my analysis is that the use of lazy build *and* build from hierarchy, rather than either technique in isolation.

Tables 6.14 and 6.15 also show how traversal performance varies as my various techniques are applied. There is essentially no difference in traversal performance when build from hierarchy is used instead of build from polygon

			conf near (n=494706) (v=13031)	conf near	conf far (n=494706) (v=156437)	conf far
Lazy	Scan	Hier	build time	trace time	build time	trace time
+	+	+	0.129 s	1.631s s	0.674 s	2.150 s
+	+	-	0.825 s	2.464s s	1.208 s	3.529 s
+	-	+	0.403 s	1.577s s	2.095 s	2.031 s
+	-	-	3.397 s	1.673s s	5.898 s	2.099 s
-	+	+	1.190 s	1.629s s	1.378 s	2.152 s
-	+	-	1.403 s	2.465s s	1.650 s	3.507 s
-	-	+	5.507 s	1.580s s	5.668 s	2.032 s
-	-	-	10.792 s	1.673s s	10.995 s	2.096 s

Table 6.15: Build times for a k-d tree with various fast-build capabilities enabled and disabled. Results are presented for two viewpoints of the Conference scene. These results were taken from the Razor system on a single core of an Intel Core 2 Duo 2.667 GHz machine. The previous table contains information about a different scene.

"soup" for the courtyard scene. Scan-based build slightly reduces trace performance as compared to a sort-based build but never by more than 2% in that scene. In the conference scene, hierarchy provides a tracing speed improvement in all configurations. The improvement stems from the fact that a good hierarchy can overcome the shortsightedness of the greedy SAH in order to produce better (more informed) trees. Most noticeable is the fact that with the near camera position, the hierarchy-based builder produces a tree that is more than 5% faster to trace than the greedy full SAH. In this scene, the scan algorithm produces noticeably poor trees without hierarchy. This is due to the scan only considering one axis for its top level splits (using the Hybrid method from the scan section).

Figure 6.11 shows how build time grows with increasing n when build from hierarchy is used without lazy build but with fast scan. The earlier theoretical results predict $O(n)$ behavior under this configuration and the experimental results match this prediction almost perfectly.

Figure 6.12 shows how build time grows with increasing v when build from hierarchy is combined with both lazy build and fast scan. The earlier theoretical results predict $O(v)$ behavior under this configuration. The experimental results roughly match this predicted behavior, but there are clearly some other effects present as well. It should be noted that it is difficult to measure the effect of varying v precisely, because it is not possible to change v without also indirectly changing other variables in the system. In this case, I varied v by choosing different viewpoints within the same model. However, different viewpoints do not behave exactly the same way even if they have the same amount of visible geometry. I also suspect that memory hierarchy effects are causing nonlinear behavior in some regions of the curve. Despite these issues it is clear that build time grows approximately linearly with v over large portions of the curve, as expected.

6.6 Concluding Remarks

This chapter presents implementation details and results for the various acceleration structures and algorithms I have presented in previous chapters. In addition I present a technical discussion demonstrating the improved effectiveness of SAH build using the correction to the surface area metric for

mailboxing. In short summary: my system using the perspective grid demonstrates higher frame rates for primary visibility and hard shadows than other fully dynamic ray tracing systems (up to 5x measured). In fact, I demonstrate that the perspective grid acceleration structure provides performance comparable to software rasterization. The perspective surface area metric is twice as effective as the original surface area metric in perspective space. In most cases, the additional cost for building perspective based acceleration structures is shown to be less than the savings provided by the faster traversal of those structures. Finally, when used in conjunction and with laziness, the two improved kd-tree build algorithms are shown to provide up to two orders of magnitude speed improvement over the traditional sort based build algorithm.

For the practical reader, I will now summarize the benefits of the various algorithms presented in this section. This list is intended to provide a concise summary of the numeric results found in this chapter.

- The correction to the SAH for mailboxing reduces intersection tests by about 30% and reduces rendering time by about 5%.
- The perspective grid system is 1x-5x faster than other dynamic ray tracing systems for eye rays. This range is wide in order to conservatively accommodate the widely different systems and architectures I compare to.
- The perspective grid system is 1x-2.5x faster than other dynamic ray tracing systems for eye and hard shadow rays.
- A perspective space kd-tree using the PSAH is 20%-40% faster than a world space kd-tree using the SAH for DoF and soft-shadow rays.
- In perspective space, the PSAH produces kd-trees that are 70%-100% faster than those produced by the SAH.
- The scan algorithm for building kd-trees is 3x-10x faster than the sort based build algorithm.
- The combined scan algorithm, build from hierarchy algorithm and lazy build is 20x-100x faster than the sort based build algorithm.

Figure 6.8: The Courtyard Scene (character models ©2003-2006 Digital Extremes, used with permission.) Models are presented in detail in the second appendix.

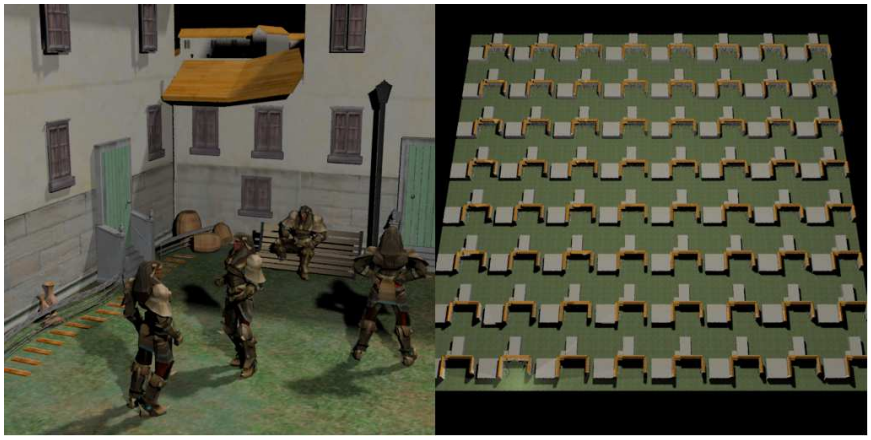

Figure 6.9: Scene (Courtyard_64) used for gathering experimental results for the build from hierarchy algorithm. In the near viewpoint shown at left, 9392 primitives are visible. In the far viewpoint shown at right, 89040 primitives are visible. It has 541023 primitives total.

Figure 6.10: Scene (Conference Room) used for gathering experimental results for the build from hierarchy algorithm. In the near viewpoint shown at left, 13031 primitives are visible. In the far viewpoint shown at right, 156437 primitives are visible. It has 494706 primitives total.

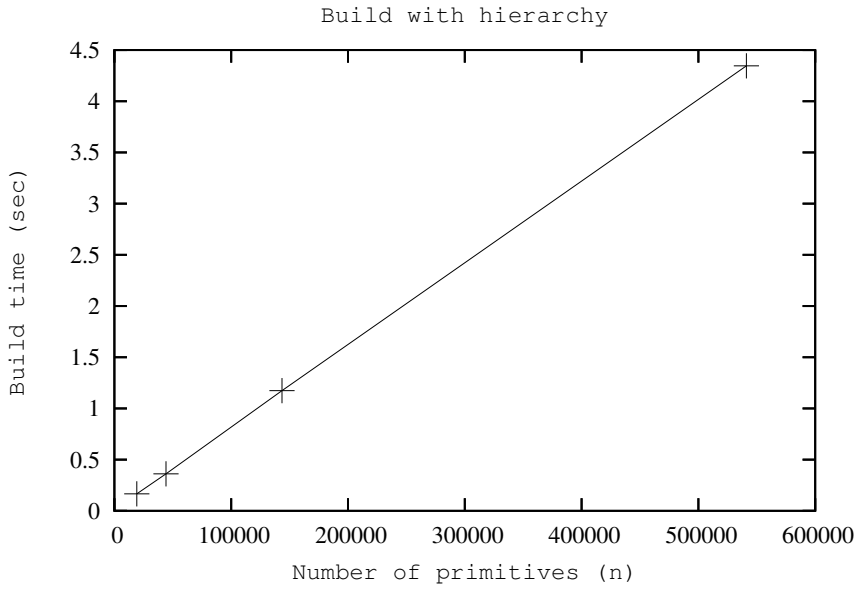

Figure 6.11: Performance of build-from-hierarchy as a function of n, the number of primitives. Lazy is disabled, and scan is enabled. We vary n by adding additional occluded geometry to the scene.

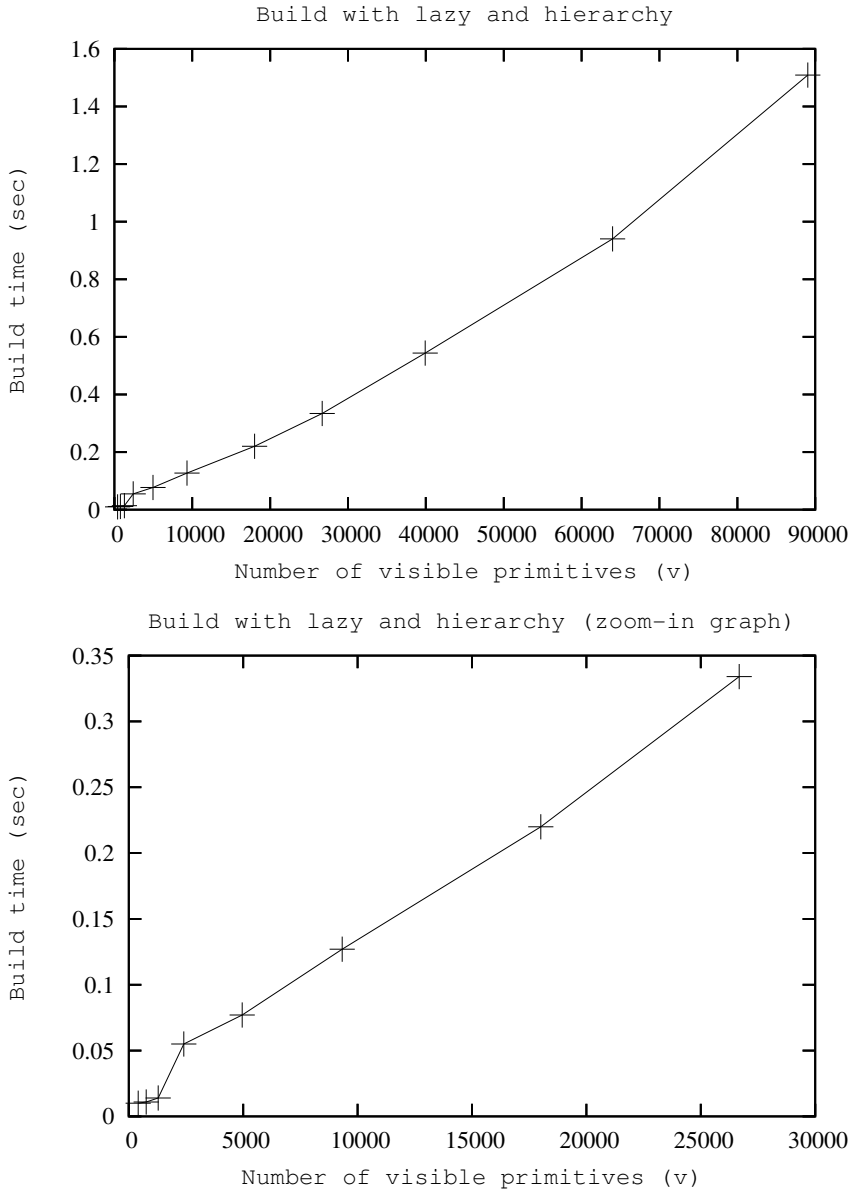

Figure 6.12: Performance of lazy build from hierarchy as a function of v, the number of visible primitives. Top: full graph, Bottom: zoom-in graph for small v.

Chapter 7

Conclusions

In this dissertation, I presented several data structures and algorithms for high-performance ray tracing. The work focuses on improving performance of visibility queries. I hypothesize that no single best acceleration structure exists and that structures specialized to specific cases will provide better performance. I supported this hypothesis by the design and implementation of specialized acceleration structures using the perspective transform. I also demonstrated performance improvements for algorithms in support of specialized acceleration structures such as high-performance build algorithms and better cost functions.

I demonstrated that these specialized acceleration structures can provide significantly better performance (lower time to image) than state-of-the-art general-purpose acceleration structures for the same workload (eye and shadow rays). I show that, in fact, these specialized acceleration structures provide a large enough reduction in traversal and intersection cost to more than make up for the additional work required to build them. I provide a high performance implementation of these data structures that demonstrates that theoretical benefits translate into practical runtime improvements.

In addition to supporting these hypotheses, I contributed a conceptual framework for classifying acceleration structures and rendering algorithms. This framework was used to justify optimizations commonly used in modern rendering systems as well as the ones I introduced. I hope this taxonomy and discussion provides a better understanding of the science of building acceleration structures some insight into why they behave the way they do.

In addition to the conceptual framework, I developed several novel cost metrics for acceleration structures. In general, cost metrics are formulae that evaluate the effectiveness of acceleration structures. The cost metrics I presented make better assumptions than do the traditional surface area metric. In particular, I derived a cost metric for perspective space acceleration structures. Since these structures are built separately for each light and camera in a scene, they can make unique assumptions about the rays that use them. I show that the new metric reduces run time in half when compared to the traditional surface area metric for perspective space structures. I also discuss cost metrics for costs other than traversal, including memory usage and the total work performed for a particular node in a hierarchical acceleration structure. Finally I present a simple correction to the surface area metrics to account for the mailboxing optimization. Mailboxing eliminates redundant intersection tests performed in partitioning based acceleration structures and in doing so affects traversal costs. This one-line correction accounts for the effects of mailboxing and provides a 30% reduction in intersection tests and a modest 5% improvement in time to image. As a byproduct, this correction

makes mailboxing a more effective optimization.

Finally, I present two novel build algorithms for surface area metric (SAM) based acceleration structures. These algorithms were initially motivated by the need to support dynamic scenes using ray tracing. In a dynamic scene, acceleration structures must be updated or rebuilt in order to support changing geometry over time. Acceleration structure build has until recently been considered an offline process and did not allow changes to the geometry. At the Siggraph'05 course, faster acceleration structure build was voted the most important problem in interactive ray tracing. The first of these two algorithms I presented was an algorithm to quickly and effectively approximate the surface area metric. Traditionally, the SAM is evaluated over a large number of candidates in order to find the lowest-cost split. Evaluating many candidates efficiently requires sorting the candidates which can be very time consuming. My algorithm samples the SAM in an adaptive manner in order to produce a low-error, piecewise quadratic approximation from which a minimum can be obtained. An implementation shows that this machine-friendly algorithm can build high quality kd-trees as much as an order of magnitude faster than the previous state-of-the-art. The second algorithm I presented uses an input hierarchy to avoid handling geometric primitives independently. By considering objects in aggregate, the build considers many fewer objects when choosing split planes thus reducing build costs. In addition to reducing actual build times, this algorithm reduces asymptotic build time over n primitives from $O(n \log n)$ to $O(n)$. Additionally, this algorithm improves the

effectiveness of demand-driven build algorithms. I demonstrated up to two orders of magnitude speed improvement over the previous state-of-the-art using all of the build optimizations I presented.

As mentioned previously, the work in this dissertation is focused on designing new data structures and algorithms for real-time ray tracing. In addition, I presented results with both theoretical and practical benefits. Much of this research has been dedicated to improving runtime performing for high quality rendering with applications ranging from video gaming to medical imaging. This work has also been driven by the hypothesis that no single structure or algorithm will achieve the highest performance for real-time visibility. I believe this dissertation effectively supports this hypothesis.

Appendices

Appendix A

Classification

In this appendix I classify several common rendering algorithms according to the taxonomy I presented in the visibility chapter. I will then use these classifications to present an optimization path that connects ray tracing to rasterization. This path provides background for the research direction I took with the perspective based acceleration structures. Specifically, the path contains the perspective grid acceleration structure and algorithm. I will begin by classifying algorithms. Recall that the taxonomy has a seven dimensional basis.

A.1 Classification

A.1.1 PBRT (kd-tree Ray Tracing)

PBRT uses ray tracing and (by default) a kd-tree based acceleration structure. In fact, it often uses many different kd-trees, one for each different object in the scene.

- Acceleration Strategy (partitioning/aggregation) : spatial partitioning (kd-tree)
- Change of Basis (many) : uses object space transforms for object specialization
- Depth (hierarchical/flat) : hierarchical (kd-tree)
- Adaptivity (adaptive/non-adaptive) : uses the SAH
- Laziness (none/fine/coarse) : none
- Streaming (samples/objects/neither) : stream samples (rays)
- Temporal Coherence (static/rebuild/refit) : rebuild (the system build an acceleration structure before rendering a frame)

A.1.2 Lazy BVH Packet Tracing

This section classifies the rendering system from Lauterbach's RT-Deform System[48]. RT-Deform exemplifies wide packet BVH tracing. It uses 64 ray packets over a lazily constructed BVH. Packet BVH tracing actually uses two acceleration structures according to the taxonomy, one for the BVH traversal and one for packet intersection.

First the BVH:

- Acceleration Strategy (partitioning/aggregation) : aggregation
- Change of Basis (many) : none
- Depth (hierarchical/flat) : hierarchical
- Adaptivity (adaptive/non-adaptive) : none/SAH (depending on configuration)
- Laziness (none/fine/coarse) : fine (acceleration structure nodes are refined on demand)
- Streaming (samples/objects/neither) : sample streaming (packets are streamed)
- Temporal Coherence (static/rebuild/refit) : rebuild/refit (the system uses a heuristic to choose when to rebuild the hierarchy as opposed to just refitting the BVH)

Ray packets are an acceleration approach that reduces the number of samples that traverse an acceleration structure. By grouping samples into aggregations, fewer samples (sample aggregations in this case) traverse the upper levels of a hierarchy. Additionally, this aggregation can be used to perform early exit "all miss" tests for the packet. If the aggregation misses an object, then all samples contained in that packet must miss the object.

- Acceleration Strategy (partitioning/aggregation) : aggregation (interval aggregation for early exit)
- Change of Basis (many) : none
- Depth (hierarchical/flat) : flat
- Adaptivity (adaptive/non-adaptive) : none
- Laziness (none/fine/coarse) : none
- Streaming (samples/objects/neither) : streams geometry across the packet
- Temporal Coherence (static/rebuild/refit) : refit (intervals are tightened as the packet thins out)

A.1.3 Grid Ray Tracing

Perhaps the simply acceleration structure for ray tracing is the uniform grid. The grid is a partitioning acceleration structure that bins samples and objects according to their spatial extent.

- Acceleration Strategy (partitioning/aggregation) : partitioning
- Change of Basis (many) : none
- Depth (hierarchical/flat) : flat
- Adaptivity (adaptive/non-adaptive) : none/grid heuristics [40]
- Laziness (none/fine/coarse) : none
- Streaming (samples/objects/neither) : samples
- Temporal Coherence (static/rebuild/refit) : rebuild

A.1.4 Perspective Grid Ray Tracing

The perspective grid is a variant of the uniform grid acceleration structure but uses the perspective transform to align samples to a common axis. The perspective grid structure is presented in a previous chapter of this dissertation.

- Acceleration Strategy (partitioning/aggregation) : partitioning
- Change of Basis (many) : perspective transform
- Depth (hierarchical/flat) : flat
- Adaptivity : none
- Laziness (none/fine/coarse) : none
- Streaming (samples/objects/neither) : samples
- Temporal Coherence (static/rebuild/refit) : rebuild

A.1.5 Z-Buffer Rendering (Rasterization)

Z-buffering is the traditional algorithm used for high performance applications. A different z-buffer must be used for each point light or camera in a scene and the algorithm doesn't easily support non-point origin samples.

- Acceleration Strategy (partitioning/aggregation) : spatial partitioning (perpendicular to the viewing direction in perspective space)
- Change of Basis (many) : uses the perspective transform to align samples to a common axis
- Depth (hierarchical/flat) : flat (uses a grid to cover the samples in perspective space)
- Adaptivity (adaptive/non-adaptive) : none
- Laziness (none/fine/coarse) : none
- Streaming (samples/objects/neither) : streams objects
- Temporal Coherence (static/rebuild/refit) : rebuild (the z-buffer is rebuilt each frame)

A.1.6 Tiled Rasterization

Tiled rasterization is a variant of traditional rasterization (z-buffering) that uses two hierarchically organized acceleration structures. It uses a low resolution perspective grid to bin geometry into "tiles" and then uses a high resolution z-buffer to render the tiles individually. In this setup the z-buffer can be considered a separate acceleration structure at the leaves of the perspective grid. I elide the specific classification for tiled rasterization because each of the two structures it uses have been previously classified (z-buffering and the perspective grid).

A.1.7 Irregular Z-Buffer

The irregular z-buffer is a primary and hard shadow visibility algorithm (similar to the perspective grid) concurrently published by Johnson [43] and Aila [4]. The algorithm is very similar to the perspective grid except that it stores samples and streams geometry, thus retaining behavior similar to the normal z-buffer but with the expanded capability of handling arbitrary common origin samples.

- Acceleration Strategy (partitioning/aggregation) : partitioning
- Change of Basis (many) : perspective transform
- Depth (hierarchical/flat) : flat
- Adaptivity (adaptive/non-adaptive) : none
- Laziness (none/fine/coarse) : none
- Streaming (samples/objects/neither) : objects
- Temporal Coherence (static/rebuild/refit) : rebuild

A.2 Optimizing Ray Tracing into Rasterization

Having classified several common visibility data structures and algorithms, I will now describe an optimization path between ray tracing and z-buffering that is made more obvious by this taxonomy. This path will contain the perspective grid structure as a step. The existence of such a path was a major source of motivation for the development of that algorithm. For the discussion, I will assume that both the ray tracing and the z-buffer algorithm will traverse the same rays. Because of restrictions on imposed by z-buffering, this implies uniformly distributed eye rays.

I will begin with traditional uniform grid ray tracing and arrive at z-buffering via a sequence of simple changes. The uniform grid acceleration structure was classified in the following way:

- Acceleration Strategy (partitioning/aggregation) : partitioning
- Change of Basis (many) : none
- Depth (hierarchical/flat) : flat
- Adaptivity (adaptive/non-adaptive) : none/grid heuristics [40]
- Laziness (none/fine/coarse) : none
- Streaming (samples/objects/neither) : samples
- Temporal Coherence (static/rebuild/refit) : rebuild

The first major change to this algorithm in the direction of z-buffering is to transform space using the perspective transform. The perspective transform also implies some amount of clipping to avoid the perspective singularity problem. The resulting structure is a uniform grid in perspective space (the perspective grid structure). This transform is the most distinct optimization on the path between ray tracing and rasterization.

Post transform, all eye rays are parallel and axis-aligned. In particular, they are aligned to the z-axis. This alignment suggests a change in the resolution of the uniform grid. It is possible to set the grid resolution equal to the ray (screen) resolution in the x and y dimensions and set the resolution in the z dimension to one. Making this modification doesn't change the data structure per se but does have the consequence that each ray has its own grid cell.

Once we have a structure with exactly one grid cell per ray, traversal of that structure is trivial. For each ray, the algorithm looks up the cell that ray maps to and tests it for intersection with each of the triangles in that cell. One additional simple transform completes the path to z-buffering: loop interchange. Instead of storing geometry and streaming samples, we may store samples and stream geometry. Since the samples map one to one with the grid cells, the grid may be stored implicitly via a formula and the ray-object intersections may be stored as the depth along the ray at which the intersection occurs. In perspective space this depth is stored as $z' = -1/z$.

An observer may also note that z-buffering uses 2-d triangle intersection

or scan conversion to render triangles. Two dimensional triangle intersection isn't actually an acceleration structure optimization; it's an optimization for triangle intersection. Although improved triangle intersection an obvious step on the path to high performance, it isn't relevant to this analysis. Thus, after all, z-buffering and ray tracing are in fact very closely related. In addition to the obvious loop interchange distinction, the only substantive difference is a simple geometric transform.

Appendix B

Scenes

In this appendix I present all of the scenes I reference throughout my dissertation in one place. Here you will find renderings of every scene used for any rendering performance result. I also provide the source of each model and its complexity in the form of polygon count.

Figure B.1: From left to right: Courtyard, Fairy Forest, Stanford Bunny, Dragon-Bunny, Conference and ERW6.

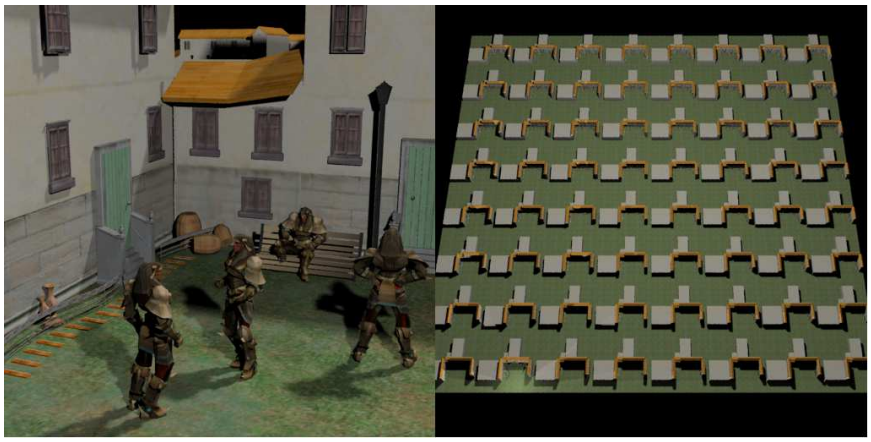

Figure B.2: Courtyard 64

Scene Title	Polygon Count	Source
Courtyard	34K	University of Texas at Austin
Courtyardx64	178k	University of Texas at Austin
Fairy Forest	174K	Daz Studios
Stanford Bunny	69K	Stanford Scanning Repository
Bunny Dragon	252K	UNC (composite of Stanford models)
Conference	282K	University of Utah
ERW6	1k	University of Utah
Hand	8K	University of Utah (poser)
Armadillo	345K	Stanford Scanning Repository
Soda Hall	1,511K	Stanford
Plants	N/A	PBRT
Sibenik	N/A	PBRT
TT	N/A	PBRT

Table B.1: Table of scenes used. Scenes with N/A polygon counts were not used in any high performance results. Scenes with no image were not used for rendering results.

Figure B.3: Conference Room (high resolution)

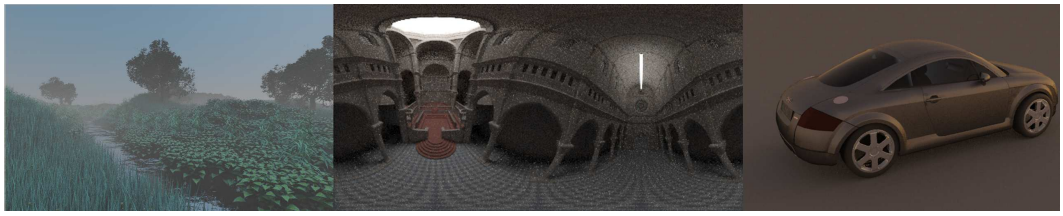

Figure B.4: From left to right: Plants, Sibenik, TT

Bibliography

[1] *Stanford 3D repository*, http://graphics.stanford.edu/data/3Dscanrep/.

[2] *Utah 3D animation repository*, http://www.sci.utah.edu/~wald/animrep.

[3] Michael Abrash, *Optimizing Pixomatic for x86 processsors: Part I*, Dr. Dobbs Journal (2004).

[4] Timo Aila and Samuli Laine, *Alias-free shadow maps*, Proceedings of Eurographics Symposium on Rendering 2004, Eurographics Association, 2004, pp. 161–166.

[5] Tomas Akenine-Mller and Timo Aila, *Conservative and tiled rasterization using a modified triangle set-up*, journal of graphics tools **10** (2005), no. 3, 1–8.

[6] James Arvo, *Ray tracing with meta-hierarchies*, In Advanced Topics in Ray Tracing, SIGGRAPH Course Notes, ACM Press, 1990, pp. 56–62.

[7] James Arvo and David Kirk, *A survey of ray tracing acceleration techniques*, An Introduction to Ray Tracing (Andrew S. Glassner, ed.), Academic Press, San Diego, CA, 1989.

[8] Carsten Benthin, *Realtime ray tracing on current cpu architectures*, Ph.D. thesis, Saarland University, Saarbrucken, Germany, January 2006.

[9] Jon Louis Bentley, *Multidimensional binary search trees used for associative searching*, Commun. ACM **18** (1975), no. 9, 509–517.

[10] J. Bigler, A. Stephens, and S. G. Parker, *Design for parallel interactive ray tracing systems*, IEEE Symp. on Interactive Ray Tracing 2006, September 2006, pp. 187–196.

[11] Solomon Boulos, Ingo Wald, and Peter Shirley, *Geometric and Arithmetic Culling Methods for Entire Ray Packets*, Tech. Report UUCS-06-010, 2006.

[12] Loren Carpenter, *The A-buffer, an antialiased hidden surface method*, SIGGRAPH Comput. Graph. **18** (1984), no. 3, 103–108.

[13] Edwin Catmull, *A subdivision algorithm for computer display of curved surfaces*, Ph.D. thesis, Dept. of CS, U. of Utah, December 1974.

[14] ———, *A hidden-surface algorithm with anti-aliasing*, SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques (New York, NY, USA), ACM, 1978, pp. 6–11.

[15] Milton Chen, Gordon Stoll, Homan Igehy, Kekoa Proudfoot, and Pat Hanrahan, *Simple models of the impact of overlap in bucket rendering*, Proc. ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware, 1998.

[16] John Cleary, Brian Wyvill, Graham Birtwistle, and Reddy Vatti, *A Parallel Ray Tracing Computer*, Proc. XI Association of Simula Users Conference. Paris (1983), 77–80.

[17] Robert L. Cook, Loren Carpenter, and Edwin Catmull, *The REYES image rendering architecture*, SIGGRAPH 87 **21** (1987), no. 4, 95–102.

[18] Robert L. Cook, Thomas Porter, and Loren Carpenter, *Distributed ray tracing*, Computer Graphics (SIGGRAPH '84 Proceedings), vol. 18, July 1984, pp. 137–45.

[19] Peter Djeu, Warren Hunt, Rui Wang, Ikrima Elhassan, Gordon Stoll, and William R. Mark, *Razor: An architecture for dynamic multiresolution ray tracing*, Tech. report, University of Texas at Austin Dep. of Comp. Science, Conditionally accepted to ACM Transactions on Graphics.

[20] Frado Durand, *Visibility, problems, techniques, and applications*, Course Notes of ACM SIGGRAPH 2000, 2000.

[21] H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, and L. Israel, *A heterogeneous multiprocessor graphics system using processor-enhanced memories*, Computer Graphics (Proc. of SIGGRAPH '89) **23** (1989), no. 3, 79–88.

[22] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor, *On visible surface generation by a priori tree structures*, SIGGRAPH '80: Proceedings of the

7th annual conference on Computer graphics and interactive techniques (New York, NY, USA), ACM, 1980, pp. 124–133.

[23] Donald Fussell and K. R. Subramanian, *Fast ray tracing using k-d trees*, Tech. report, The University Of Texas at Austin, 1988.

[24] Andrew Glassner, *Space subdivision for fast ray tracing*, IEEE Computer Graphics and Applications **4** (1984), no. 10, 15–22.

[25] Jeffrey Goldsmith and John Salmon, *Automatic creation of object hierarchies for ray tracing*, IEEE Computer Graphics and Applications **7** (1987), no. 5, 14–20.

[26] Ned Greene, Michael Kass, and Gavin Miller, *Hierarchical Z-buffer visibility*, SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques (New York, NY, USA), ACM, 1993, pp. 231–238.

[27] Johannes Günther, Heiko Friedrich, Ingo Wald, Hans-Peter Seidel, and Philipp Slusallek, *Ray tracing animated scenes using motion decomposition*, Computer Graphics Forum **25** (2006), no. 3, 517–525, (Proceedings of Eurographics).

[28] Eric Haines and Donald P. Greenberg, *The light buffer: A shadow-testing accelerator*, IEEE Computer Graphics and Applications **6** (1986), no. 9, 6–16.

[29] Vlastimil Havran, *Heuristic ray shooting algorithms*, Ph.D. thesis, Czech Technical University, Nov. 2000.

[30] ———, *Mailboxing, yea or nay?*, Ray Tracing News **15** (2002), no. 1.

[31] Vlastimil Havran and Jirí Bittner, *On improving KD trees for ray shooting*, Proceedings of WSCG, 2002, pp. 209–216.

[32] Paul S. Heckbert and Pat Hanrahan, *Beam tracing polygonal objects*, SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques (New York, NY, USA), ACM, 1984, pp. 119–127.

[33] Emile Hsieh, Vladimir Pentkovski, and Thomas Piazza, *ZR: a 3D API transparent technology for chunk rendering*, MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture (Washington, DC, USA), IEEE Computer Society, 2001, pp. 284–291.

[34] Warren Hunt, *Corrections to the surface area metric with respect to mailboxing*, IEEE Symposium on Interactive Ray Tracing 2008, IEEE, 2008.

[35] Warren Hunt and William Mark, *Adaptive acceleration structures in perspective space*, IEEE Symposium on Interactive Ray Tracing 2008, IEEE, 2008.

[36] ———, *Ray-specialized acceleration structures for ray tracing*, IEEE Symposium on Interactive Ray Tracing 2008, IEEE, 2008.

[37] Warren Hunt, William Mark, and Don Fussell, *Fast and lazy build of acceleration structures from scene hierarchies*, 2007 IEEE Symposium on Interactive Ray Tracing, IEEE, Sept. 2007, pp. 47–54.

[38] Warren Hunt, William Mark, and Gordon Stoll, *Fast kd-tree construction with an adaptive error-bounded heuristic*, Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, 2006, pp. 81–88.

[39] James T. Hurley, Alexander Kapustin, Alexander Reshetov, and Alexei Soupikov, *Fast ray tracing for modern general purpose CPU*, Proceedings of GraphiCon, 2002.

[40] Thiago Ize, Ingo Wald, and Steven G. Parker, *Grid creation strategies for efficient ray tracing*, Proceedings of the 2007 IEEE/Eurographics Symposium on Interactive Ray Tracing, 2007, pp. 27–32.

[41] Thiago Ize, Ingo Wald, Chelsea Robertson, and Steven G. Parker, *An evaluation of parallel grid construction for ray tracing dynamic scenes*, Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, 2006, pp. 47–55.

[42] Chris L. Jackins and Steven L. Tanimoto, *Oct-trees and their use in representing three-dimensional objects*, Computer Graphics and Image Processing **14** (1980), no. 3, 249–270.

[43] Gregory S. Johnson, Juhyun Lee, Christopher A. Burns, and William R. Mark, *The irregular Z-buffer: Hardware acceleration for irregular data*

structures, ACM Trans. Graph. **24** (2005), no. 4, 1462–1482.

[44] Norman P. Jouppi and Chun-Fa Chang, *Z3: an economical hardware technique for high-quality antialiasing and transparency*, HWWS '99: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware (New York, NY, USA), ACM, 1999, pp. 85–93.

[45] Alan Kapler, *Evolution of a vfx voxel tool*, SIGGRAPH '02: ACM SIGGRAPH 2002 conference abstracts and applications (New York, NY, USA), ACM, 2002, pp. 179–179.

[46] Douglas Scott Kay and Donald Greenberg, *Transparency for computer synthesized images*, SIGGRAPH '79: Proceedings of the 6th annual conference on Computer graphics and interactive techniques (New York, NY, USA), ACM, 1979, pp. 158–164.

[47] Samuli Laine, Timo Aila, Ulf Assarsson, Jaakko Lehtinen, and Tomas Akenine-Möller, *Soft shadow volumes for ray tracing*, ACM Transactions on Graphics **24** (2005), no. 3, 1156–1165.

[48] Christian Lauterbach, Sung-Eui Yoon, David Tuft, and Dinesh Manocha, *RT-DEFORM: Interactive ray tracing of dynamic scenes using BVHs*, Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, 2006, pp. 39–45.

[49] J. David MacDonald and Kellogg S. Booth, *Heuristics for ray tracing using space subdivision*, Visual Computer **6** (1990), no. 6, 153–65.

[50] William Martin, Erik Reinhard, Peter Shirley, Steven Parker, and William Thompson, *Temporally coherent interactive ray tracing*, Journal of Graphics Tools **2** (2001), 41–48.

[51] Alexei Soupikov Maxim Shevtsov and Alexander Kapustin, *Ray-triangle intersection algorithm for modern cpu architectures*, Proceedings of International Conference on Computer Graphics & Vision, 2007.

[52] Gene S. Miller and C. Robert Hoffman, *Illumination and reflection maps: Simulated objects in simulated and real environments*, 1984.

[53] Steve Molnar, Michael Cox, David Ellsworth, and Henry Fuchs, *A sorting classification of parallel rendering*, IEEE computer graphics and applications (1994), no. 4, 23–32.

[54] Matt Pharr and Greg Humpreys, *Physically based rendering: From theory to implementation*, Morgan Kaufmann, 2004.

[55] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek, *Experiences with streaming construction of SAH kd-trees*, Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, 2006.

[56] RAD Game Tools, *Pixomatic SDK Features*, visited Jan 18, 2008.

[57] Alexander Reshetov, *Faster ray packets - triangle intersection through vertex culling*, 2007 IEEE Symposium on Interactive Ray Tracing, IEEE, Sept. 2007, pp. 105–112.

[58] Alexander Reshetov, Alexei Soupikov, and Jim Hurley, *Multi-level ray tracing algorithm*, SIGGRAPH '05: ACM SIGGRAPH 2005 Papers (New York, NY, USA), ACM, 2005, pp. 1176–1185.

[59] David Roger, Ulf Assarsson, and Nicolas Holzschuch, *Whitted ray-tracing for dynamic scenes using a ray-space hierarchy on the gpu*, Rendering Techniques 2007 (Proceedings of the Eurographics Symposium on Rendering) (Jan Kautz and Sumanta Pattanaik, eds.), Eurographics and ACM/SIGGRAPH, the Eurographics Association, June 2007, pp. 99–110.

[60] Steve Rubin and Turner Whitted, *A 3D representation for fast rendering of complex scenes*, Proceedings of SIGGRAPH, 1980, pp. 110–116.

[61] Steven Rubin and Turner Whitted, *A 3-dimensional representation for fast rendering of complex scenes*, SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques (New York, NY, USA), ACM, 1980, pp. 110–116.

[62] David Salesin and Jorge Stolfi, *The ZZ-buffer: A simple and efficient rendering algorithm with reliable antialiasing*, Proceedings of the PIXIM '89 Conference (Hermes Editions, Paris, France), 1989, pp. 451–66.

[63] ———, *Rendering CSG models with a ZZ-buffer*, SIGGRAPH Comput. Graph. **24** (1990), no. 4, 67–76.

[64] Andreas Schilling and Wolfgang Strasser, *EXACT: algorithm and hardware architecture for an improved A-buffer*, SIGGRAPH '93: Proceedings

of the 20th annual conference on Computer graphics and interactive techniques (New York, NY, USA), ACM, 1993, pp. 85–91.

[65] Jonathan Shade, Steven Gortler, Li wei He, and Richard Szeliski, *Layered depth images*, SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques (New York, NY, USA), ACM, 1998, pp. 231–242.

[66] Oliver Dvel Stefan Zerbst, *3d game engine programming*, Thomson Course Technology, 2004.

[67] Ingo Wald, Solomon Boulos, and Peter Shirley, *Ray tracing deformable scenes using dynamic bounding volume hierarchies*, ACM Transactions on Graphics **26** (2007), no. 1, 1–18.

[68] ———, *Ray tracing deformable scenes using dynamic bounding volume hierarchies*, ACM Trans. Graph. **26** (2007), no. 1, 6.

[69] Ingo Wald and Vlastimil Havran, *On building fast kd-trees for ray tracing, and on doing that in O(N log N)*, Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, 2006, pp. 61–70.

[70] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G. Parker, *Ray tracing animated scenes using coherent grid traversal*, ACM Transactions on Graphics **25** (2006), no. 3, 485–493, (Proceedings of ACM SIGGRAPH).

[71] Ingo Wald, William R. Mark, Johannes Günther, Solomon Boulos, Thiago Ize, Warren Hunt, Steven G. Parker, and Peter Shirley, *State of the art in ray tracing animated scenes*, Eurographics 2007 State of the Art Reports, Eurographics Association, 2007.

[72] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner, *Interactive rendering with coherent ray tracing*, Proc. of Eurographics 2001, 2001.

[73] Yulan Wang and Steven Molnar, *Second-depth shadow mapping*, Tech. report, Chapel Hill, NC, USA, 1994.

[74] Hank Weghorst, Gary Hooper, and Donald P. Greenberg, *Improved computational methods for ray tracing*, ACM Trans. Graph. **3** (1984), no. 1, 52–69.

[75] Turner Whitted, *An improved illumination model for shaded display*, Communications of the ACM **23** (1980), no. 6, 343–349.

[76] Lance Williams, *Casting curved shadows on curved surfaces*, SIGGRAPH Comput. Graph. **12** (1978), no. 3, 270–274.

[77] Andrew Woo, *The shadow depth map revisited*, (1992), 338–342.

[78] Andrew Woo, Andrew Pearce, and Marc Ouellette, *It's really not a rendering bug, you see ...*, IEEE Comput. Graph. Appl. **16** (1996), no. 5, 21–25.

[79] Sung-Eui Yoon, Sean Curtis, and Dinesh Manocha, *Ray tracing dynamic scenes using selective restructuring*, SIGGRAPH '07: ACM SIGGRAPH 2007 sketches (New York, NY, USA), ACM, 2007, p. 55.

Index

Vita

Warren Andrew Hunt was born in Austin, Texas on 10 March 1983, the son of Dr. Warren A. Hunt Jr. and Irene R. Hunt. He recieved a Bachelor of Science degree in Computer Science and another in Computational and Applied Mathematics from Carnegie Mellon University in 2004. He began his graduate studies at the University of Texas at Austin in fall of that year.

Permanent address: 2106 Ringtail Ridge
Austin, Texas 78746

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.