EFFICIENT ACCELERATION STRUCTURES FOR RAY TRACING STATIC AND DYNAMIC SCENES

by

Thiago Ize

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

August 2009

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Thiago Ize

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

	Chair: Steven G. Parker
	Ingo Wald
	Peter Shirley
	Claudio Silva
	Alexander Reshetov

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the dissertation of Thiago Ize in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date	Steven G. Parker Chair, Supervisory Committee

Approved for the Major Department

Martin Berzins
Chair/Dean

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

This dissertation presents efficient acceleration structures for ray tracing static and dynamic scenes. The first method for reducing render time shows how the general binary space partitioning (BSP) tree, unlike the kd-tree variant that uses axis aligned splitting planes, can be used to render static scenes roughly as fast to several times faster than other acceleration structures. The BSP is especially useful for handling geometry configurations that cause substantial slow downs in other acceleration structures; however, its slow build time limits it to applications where the scene can be built offline. Another method improves grid-based acceleration structure efficiency by analytically determining the optimal single or multilevel grid resolutions for various classes of scenes. In particular, we show for manifold-like models composed of compact triangles that storage is linear in the number of triangles, N, and render complexity goes from $O(N^{1/3})$ for single level grids to $O(N^{1/(2d+1)})$ for d levels. On the other hand, manifold-like models composed of long skinny triangles require super linear memory in order to achieve sublinear time complexity; for a single level grid, our analysis states that $O(N^{3/2})$ cells must be used and that this gives a time complexity of $O(N^{1/2})$. If coherent ray packets are available, we show that using a frustum traversal allows us to simultaneously traverse a packet of rays through a grid, which results in a substantial speedup over the traditional single ray traversal.

Dynamic scenes require that the acceleration structure be quickly updated each frame. Pairing the frustum traversal with a quick grid rebuild allows for interactive ray tracing of fully dynamic scenes. Furthermore, we present a highly efficient parallel grid build that scales to many processors. For dynamic scenes that are deformable, a bounding volume hierarchy (BVH) can be updated by quickly refitting its bounding volumes rather than performing a slow BVH rebuild. Unfortunately, refitting can degrade the quality of the tree and lead to worse render times. We maintain a high quality BVH without having to wait for the slow rebuild by refitting the BVH every frame while simultaneously building another BVH from scratch, possibly over the course of several frames. When the new BVH is complete, it replaces the older refitted BVH.

CONTENTS

ABSTRACT . . . . **iv**

LIST OF FIGURES . . . . **ix**

LIST OF TABLES . . . . **xi**

NOTATION AND SYMBOLS . . . . **xii**

ACKNOWLEDGEMENTS . . . . **xiii**

CHAPTERS

1. INTRODUCTION **1**

1.1 Real-Time Ray Tracing and Dynamic Scenes 2
1.2 Thesis 3
1.3 Publications 3

2. BACKGROUND **4**

2.1 Acceleration Structures 4
2.1.1 Grids 4
2.1.2 Binary Space Partitioning Trees 5
2.1.3 Bounding Volume Hierarchies 6
2.1.4 Surface Area Heuristic 6
2.2 Interactive Ray Tracing 7
2.3 Dynamic Scenes 8
2.3.1 Grids 8
2.3.2 Bounding Volume Hierarchies 9
2.3.3 Binary Space Partitioning Trees 9

3. RAY TRACING WITH THE REAL BSP **10**

3.1 Introduction 10
3.2 Background 11
3.2.1 Binary Space Partitioning Trees 11
3.3 Building the BSP 12
3.3.1 Intersection of Half-Spaces 12
3.3.2 Handling Numerical Precision 13
3.3.3 BSP Surface Area Heuristic 13
3.3.4 Which Planes to Test 14
3.3.5 Build 14
3.4 Ray Tracing Using Generalized BSPs 15
3.4.1 Traversal 15
3.4.2 Packet Traversal 16

3.4.3 Traversal-based Triangle Intersection 17
3.4.4 Data Structure 17
3.5 Results 17
3.5.1 Statistical Comparison 21
3.5.2 Absolute Performance Comparison 25
3.5.3 Build Time and Build Efficiency 27
3.6 Summary and Discussion 27
3.6.1 Future Work 30

4. GRID CREATION STRATEGIES FOR EFFICIENT RAY TRACING 33

4.1 Introduction 33
4.2 Classic Results for One-Level Grids 34
4.2.1 Theoretical Analysis 35
4.2.2 Empirical Test of Analysis 35
4.3 Long-Skinny Triangles and One-Level Grids 36
4.3.1 Theoretical Analysis 36
4.3.2 Empirical Test of Analysis 38
4.4 Multilevel Grids and Points on Surfaces 40
4.4.1 Theoretical Analysis for 2-Level Grids 41
4.4.2 Empirical Tests for 2-Level Grids 42
4.4.3 Theoretical Analysis for Multilevel Grids 43
4.4.4 Empirical Tests for Multilevel Grids 45
4.4.5 Two Level Grids and Long-Skinny Triangles 46
4.5 Conclusion and Discussion 46

5. RAY TRACING FULLY DYNAMIC SCENES WITH THE COHERENT GRID TRAVERSAL 51

5.1 Introduction 51
5.2 Coherent Grid Traversal 51
5.2.1 Issues with Packetized Grids 51
5.2.2 A Slice-based Packet Traversal for Grids 52
5.2.2.1 Extension to frustum traversal. 54
5.2.2.2 Traversal setup. 55
5.2.2.3 Incremental traversal. 55
5.2.3 Efficient Slice and Triangle Intersection 55
5.2.3.1 SIMD frustum culling. 56
5.2.3.2 Mailboxing. 56
5.2.3.3 Impact of mailboxing and frustum culling. 57
5.2.4 Extension to Hierarchical Grids 58
5.3 Acceleration Structure Rebuild 59
5.4 Experiments and Results 61
5.4.1 Impact of Grid and Packet Resolution 62
5.4.2 Scalability with Screen Resolution 63
5.4.3 Performance for Static Scenes 64
5.4.4 Scalability with Model Resolution 64
5.4.5 Comparison to Single-Ray Grid Traversal 65
5.4.6 Performance for Animated Scenes 66
5.4.6.1 Animated meshes. 67

5.4.6.2 Non-hierarchical animation. 68
5.4.6.3 A real-world example. 68
5.4.7 Shading, Shadows, and Secondary Rays 70
5.5 Summary and Discussion 71

6. AN EVALUATION OF PARALLEL GRID CONSTRUCTION FOR RAY TRACING DYNAMIC SCENES 74

6.1 Introduction 74
6.2 Background 75
6.2.1 Dynamic Scenes and CGT 75
6.2.2 Parallel Computers 75
6.2.2.1 Multicore architectures. 75
6.2.2.2 Distributed memory, SMP, and NUMA. 76
6.2.3 Parallel Construction of Data Structures 76
6.3 Parallel Grid Rebuild 77
6.3.1 Parallel Cell Clearing and Macrocell Build 77
6.3.2 Parallel Triangle Insertion 78
6.3.2.1 Sort-first build. 78
6.3.2.2 Sort-last build. 78
6.3.2.3 Sort-middle build. 79
6.3.3 Sort-Last Approaches 79
6.3.3.1 Single mutex. 80
6.3.3.2 One mutex per cell. 80
6.3.3.3 Mutex pool. 80
6.3.3.4 Grid merge. 81
6.3.4 Sort-Middle Grid Build 81
6.4 Results 83
6.4.1 Comparison of Build Methods 83
6.4.2 Individual Steps 85
6.4.3 Memory System Overhead 86
6.4.3.1 NUMA. 87
6.4.4 Comparison Using Other Scenes 89
6.4.5 Overall Parallel Rendering Performance 90
6.5 Summary and Discussion 93

7. FAST, PARALLEL, AND ASYNCHRONOUS CONSTRUCTION OF BVHS FOR RAY TRACING ANIMATED SCENES 95

7.1 Introduction 95
7.2 Background 96
7.2.1 Handling BVH Deterioration 96
7.2.2 Fast Construction of SAH Hierarchies 97
7.3 System Overview 97
7.4 Fast Construction of SAH-based BVHs 98
7.5 Parallel Construction of SAH-based BVHs 99
7.5.1 Results 101
7.6 Asynchronous Dynamic BVHs 102
7.6.1 Parallel Rendering and Update Steps 104
7.6.1.1 Vertex generation. 104

7.6.1.2 Parallel BVH refit. 104
7.6.1.3 Triangle update. 104
7.6.1.4 Parallel rendering. 105
7.6.1.5 BVH build method. 105
7.7 Results and Discussion 105
7.7.1 Comparison to Synchronous Approaches 106
7.7.2 Fast vs. Slow BVH Building 108
7.7.3 Relation on Build Time and Frame Rate 110
7.7.4 Number of Threads and Choice of Build Method 110
7.7.5 Remaining Limitations 111
7.8 Conclusion 112

8. CONCLUSION 113

8.1 Static Scenes 113
8.2 Dynamic Scenes 115
8.3 Future Work 115

REFERENCES 117

LIST OF FIGURES

3.1 Path traced architectural scenes 19

3.2 Ray cast and path traced scenes 20

3.3 Rendering time visualization 28

4.1 2D versions of the scene types analyzed. 34

4.2 Fitted runtimes. 37

4.3 Four of the models we use in our tests 39

4.4 Multilevel grid for points 40

4.5 Two different ways to refine a cell. 41

4.6 Rendering time visualization for 2-level grids 44

4.7 Multilevel grid cost validation 47

5.1 Ray divergence in a grid 53

5.2 Coherent grid traversal 54

5.3 Frustum culling 56

5.4 Mailboxing 57

5.5 λ's affect on overall framerate 61

5.6 Static render time with varying packet sizes and different resolutions of the Stanford Armadillo 63

5.7 Architectural scenes 64

5.8 Performance with varying model resolution 66

5.9 Simple animated scenes 68

5.10 Complex animated scenes 69

6.1 Sort-middle build 82

6.2 Total rebuild performance 84

6.3 Individual build steps 85

6.4 Simulation of ideal parallel computer 87

6.5 Interleaving memory 88

6.6 Test scenes 90

6.7 Effect of triangle distribution 91

6.8 Overall scalability 92
7.1 Asynchronous framework 103
7.2 Test scenes 106
7.3 Impact on frame rate using various build strategies 107
7.4 Impact of build time in the asynchronous system 109

LIST OF TABLES

3.1 Per ray statistics 23

3.2 Rendering performance 25

3.3 Build statistics. 26

4.1 M accuracy: one-level grid. 38

4.2 M accuracy: one-level grid for long-skinny triangles. 39

4.3 M accuracy: 2-level grid 43

5.1 Ray-triangle intersection tests. 58

5.2 Comparison with other acceleration structures 65

5.3 Static scene performance 66

5.4 Grid statistics based on ray packet size 67

6.1 Rebuild scalability 91

7.1 Build and render performance of various builds 100

7.2 Build scalability 101

NOTATION AND SYMBOLS

symbol	meaning
N	number of geometric primitives
M	number of grid cells
M_i	number of grid cells at level i ($i = 1$ is the top)
m	number of grid cells in one dimension: $m = \sqrt[3]{M}$
m_i	the cube root of M_i
k	a grid has km^2 occupied cells
T	average time to trace a single random ray
T_{setup}	time to do initial intersection with scene bounding box and setup traversal
T_{step}	time for ray to advance from cell to its neighbor
$T_{\text{intersection}}$	ray/primitive intersection time

ACKNOWLEDGEMENTS

The Bunny, Happy Buddha, Thai Statue, and Lucy models are courtesy of the Stanford Scanning Repository. The Exploding Dragon is courtesy of the UNC dynamic scene Benchmarks. The truck model is courtesy of the US Army Research Laboratory. The conference room was created by Anat Grynberg and Greg Ward.

I would like to thank Justin Luitjens for his insightful suggestions for the parallel grid build analysis and Peter Shirley for his suggestions on the BSP tree. Most importantly, I owe a sincere thank you to Ingo Wald and Peter Shirley for their substantial contributions as coauthors to the papers that make up this dissertation. Finally, Steve Parker for funding me, advising me, helping on all my papers, and most importantly, shielding me from all the needless time sinks so that I would have more free time to pursue the work I love.

CHAPTER 1

INTRODUCTION

The past decade has seen interactive ray tracing go from an infeasible dream to only being feasible on a million dollar supercomputer to running on a laptop. Furthermore, along with increasing frame rates, the domain of interactive ray tracing has grown to encompass fully dynamic scenes and more advanced shading effects. This progress is due to faster hardware, better optimized code, and superior algorithms.

Ray tracing and rasterization are used in computer graphics to generate images of a virtual scene. Ray tracing generates an image by simulating how light interacts with the scene and camera and is therefore easily capable of producing highly realistic images if enough light photons are simulated. Rasterization on the other hand is only able to specify which objects are visible to the camera, but not how they should appear with respect to the other objects in the scene. This precludes rasterization from directly handling effects such as shadows, reflections, and indirect lighting. By using clever tricks and approximations, rasterization is able to simulate these effects; however, the code complexity and time to render a frame increase, while visual errors, which are sometimes quite noticeable, are introduced. Both techniques are used in the film industry where interactive rendering is not required [14]; however, video games currently only render using rasterization since dedicated special purpose hardware called a graphics processing unit (GPU) allows GPU accelerated rasterization for video games to be performed much faster than ray tracing using a conventional processor (CPU). When high quality ray tracing can be made interactive, video games will also be able to use ray tracing in order to achieve more realistic images.

Ray tracing simulates light by tracing a ray of light from a light source, calculating which objects it reflects and refracts off of, and finally whether it hits the camera and can be recorded. Since we normally are only interested in rays of light that end up hitting the camera and so contribute to the image, a common and trivial optimization is to reverse the path of light and have it start from the camera and end at the light source; the physics and math end up being the same. The other very common optimization is to use an acceleration structure to aid in determining which object a ray hits without having to test the ray with every object in the scene.

The two most expensive steps in ray tracing are usually in traversing rays through an acceleration structure and performing the ray-primitive intersection tests. For this reason, substantial gains in overall ray tracing time are usually achieved by making these two steps faster. The ray-primitive intersection testing can be made faster through more efficient intersection tests or by lowering the number of tests required. Acceleration structures are made faster by making traversal steps faster or by requiring less traversal steps. Unfortunately, time reductions in one place can often result in increases in others, thereby making it a challenge to achieve an overall speedup.

One very effective way of achieving a speedup in both intersection and traversal is by utilizing single instruction multiple data (SIMD) units on modern processors. In the case of 4-wide SIMD units, this has effectively allowed for up to 4× speedups in both ray-primitive intersection and ray traversal. The most common way to take advantage of SIMD is to create a ray packet that contains multiple rays and then simultaneously traverse them through the structure as well as intersecting the packet of rays against individual primitives, all using SIMD.

Another major advance that logically builds on top of SIMD algorithms is to use a proxy during traversal so that instead of computing how each ray traverses a structure, a single proxy can be substituted for many, if not all, of the rays in the packet, thereby allowing the traversal of all the rays to be accomplished by traversing a few or even only one object. A commonly used proxy is to traverse a frustum that bounds all the rays in the packet [67, 80]. The other is to use a subset of the rays in the ray packet to conservatively represent how all the rays might traverse the structure [78]. Unfortunately, these techniques often behave poorly when the ray packet is composed of incoherent rays, such as occurs in path tracing.

1.1 Real-Time Ray Tracing and Dynamic Scenes

Fast ray tracing depends on using efficient spatial acceleration structures, such as a bounding volume hierarchy (BVH), kd-tree, or grid. Whereas there has been a long running debate on which of these is best, by 2005, virtually all fast ray tracers were built on kd-trees. Unfortunately, kd-trees are costly to build and cannot easily be incrementally updated, and as such, present an obstacle to handling dynamic scenes. Thus, researchers again started actively exploring better ways to support dynamic scenes with other acceleration structures [82].

There are two common ways to support dynamic scenes. The simplest and most robust is to rebuild the acceleration structure whenever the scene changes. This, however, can be prohibitively expensive since many acceleration structures cannot be rebuilt at interactive rates. The other option is to only update parts of the acceleration structure, although to be efficient, this might place restrictions on the type of animation that can be allowed.

As a result of these efforts, ray tracing animated scenes has been demonstrated using both kd-trees [23, 69], grids [80], and BVHs [52, 78, 88], with various performance vs. flexibility trade-offs depending on which data structure is being used. All these structures are capable of using SIMD and some sort of proxy traversal method. Generally speaking, grids are currently considered to be fastest to build but somewhat less efficient for traversal/intersection, kd-trees to be the most efficient for traversal/intersection but most costly to build, and BVHs somewhere in between in build time, and close to kd-trees for traversal.

1.2 Thesis

Substantial performance improvements in interactive ray tracing can be obtained by using a general BSP with a high quality build algorithm, using a few simple assumptions to analytically find the optimal single or multilevel grid resolutions, or traversing a packet of rays through a grid using a SIMD based frustum traversal algorithm. Furthermore, for dynamic scenes, substantial improvements can be obtained by using an extremely fast macrocell grid rebuild for each frame, by parallelizing the grid rebuild, or by rebuilding a BVH structure asynchronously to the rendering and refitting.

1.3 Publications

The results in this thesis have previously appeared in the following publications [38, 39, 40, 41, 80, 81].

CHAPTER 2

BACKGROUND

2.1 Acceleration Structures

The fundamental ray tracing kernel is to cast a ray, often from a pixel or off an object, and determine which primitive is first hit by the ray. The naive approach towards determining the hit primitive is to simply test every primitive in the scene against the ray. Of course, this results in a linear time complexity algorithm in the number of primitives and is extremely expensive. For this reason, essentially all ray tracers, including the very first ray tracer that supported more than just ray casting [86], use acceleration structures in order to achieve sublinear—usually logarithmic—time complexities.

High performance ray tracing usually requires a spatial and/or hierarchical data structure for efficiently searching the primitives in the scene. There are currently three types of acceleration structures used in modern ray tracers—uniform grids [80], binary space partitioning trees [67, 23], and bounding volume hierarchies (BVH) [52, 78]. Other types of acceleration structures, such as octrees [29] or Arvo and Kirk's ray classification [4] structure, have been proposed; however, they are currently not seen as competitive for general ray tracing of surfaces and so will not be further discussed. Acceleration structures can be further accelerated by tracing packets of coherent rays, using SIMD extensions on modern CPUs [75], and by using a proxy, such as the bounding frustum [67, 80] or interval arithmetic (IA) [78] to traverse the packet through the acceleration structure by using only a small subset of the rays in the packet. However, it is unclear how well SIMD, packet, and frustum techniques will perform for secondary rays and so it is important to still look into what the best single ray acceleration structure can achieve.

2.1.1 Grids

A uniform grid [2, 16, 28] subdivides a scene into equally sized cells, or voxels. Determining which objects a ray hits can then be quickly computed by determining which cells a ray passes through and then only checking the objects inside of those traversed cells. If cells are checked in the order that a ray traverses them, then once a ray is found to hit the objects in that cell, the traversal through the grid can terminate since the closest intersecting object has been found. Grid traversal has been further accelerated through the use of multilevel, recursive, or hierarchical grids [43, 50] and, as shown in Chapter 5, by simultaneously traversing a packet of rays through the grid by traversing a frustum in place of the individual rays [80].

2.1.2 Binary Space Partitioning Trees

One of the most fundamental concepts in these data structures is "binary space partitioning"—successively subdividing a scene with planes until certain termination criteria are reached. The resulting data structure is called a binary space partitioning tree or BSP tree.

The flexibility to place splitting planes where they are most effective allows BSP trees to adapt very well even to complex scenes and highly uneven scene distributions, usually making them highly effective. For a polygonal scene, a BSP tree can, for example, place its splitting planes exactly through the scene's polygons, thus exactly enclosing the scene. In addition, their binary nature makes them well suited to hierarchical divide and conquer and branch and bound style traversal algorithms that in practice are usually quite competitive with other algorithms. As with grids, a BSP can make use of early ray termination, meaning that once an object is hit, no other kd-tree nodes need be checked. Thus, in theory, BSP trees are simple, elegant, flexible, and highly efficient.

However, "real" BSP trees with arbitrarily oriented planes are used very rarely, if at all, in practice. In collision detection, simple BSP ray shooting queries [3] do exist, but if applied to generating images with ray tracing, would perform quite poorly. In ray tracing, we are not aware of a single high performance ray tracer that uses arbitrarily oriented planes. Instead, ray tracers typically use kd-trees [42], which are a restricted type of BSP tree in which only axis-aligned splitting planes are allowed[1]. Kd-trees provide storage and computation advantages but do not conform as well to scene geometry.

Since kd-trees allow only axis-aligned splitting planes, the traversal can be made extremely efficient and substantially faster than that of a general BSP. However, unlike an optimal BSP, an optimal kd-tree will not be able to partition all the objects into

[1]Somewhat surprisingly, Although there are many papers on ray tracing with "BSP trees" ([34, 44, 72] to name just a few), *none* of the papers we found actually uses arbitrarily oriented planes!

individual leaves since axis-aligned splits might not exist that cleanly partition triangles. This results in more ray-object intersection tests (albeit potentially fewer traversal steps) that must be performed. The prevailing belief has been that the faster traversal of the kd-tree makes up for the increase in intersection tests.

2.1.3 Bounding Volume Hierarchies

Instead of subdividing space into "voxels" of triangles, as in grids and kd-trees, BVHs [68] build an *object hierarchy*, and in each tree node store a bounding volume for that subtree's geometry. The bounding volume of choice in essentially all modern ray tracers are axis aligned boxes. Unlike grids and BSPs, which are spatial data structures, a BVH might have two child nodes that overlap and so cannot as easily terminate a ray early in the traversal.

2.1.4 Surface Area Heuristic

When building an adaptive data structure like a BSP tree, kd-tree, or BVH, the actual way that a node gets partitioned into two halves can have a profound impact on the number of expected traversal steps and primitive intersections. For these tree based structures, the best known method to build them with minimal expected cost is the greedy surface area heuristic (SAH). The greedy SAH estimates the cost of a split plane by assuming that each of the two resulting halves l and r would become leaves. Then, using geometric probability (see [31] for a more detailed explanation) the expected cost of splitting node p is

$$C_p = \frac{SA(v_l)}{SA(v_p)} n_l C_i + \frac{SA(v_r)}{SA(v_p)} n_r C_i + C_t$$

where $SA()$ is the surface area, n is the number of primitives, C_i is the cost of intersecting a primitive, and C_t is the cost of traversing a node. Though originally invented for BVHs and commonly used for kd-trees, Kammaje and Mora recently showed how it can also be applied to restricted BSPs [44] and the same theory works for BSP trees as well.

Wald and Havran showed that an SAH kd-tree can be built in $O(n \log n)$, although not at interactive rates [79]. Independently, Hunt et al. [37] and Popov et al. [64] showed that the SAH kd-tree build can be made faster by approximating the SAH cost with a subset of the candidate splits at the expense of a slight hit in rendering performance.

2.2 Interactive Ray Tracing

Before 1999, several researchers attempted to achieve interactive ray tracing systems using supercomputers [46, 60]; however, a combination of insufficient compute power and algorithmic advancements prevented them from achieving more than one frame per second. One of the first truly interactive ray tracers was implemented by Parker et al. in 1999 on a 64 node SGI Origin 2000 Reality Monster [62]. It was able to render a complex static scene at 512×512 pixels with 60 CPUs at 4 frames per second. Their design emphasized algorithmic simplicity, cache coherency optimizations, and efficient dynamic load balancing, paired with substantial computing power.

Two years later, Wald et al. developed an interactive ray tracer that could render scenes at a few frames per second on a single dual processor PC or faster using a cluster of PCs [84]. In addition to faster hardware and a somewhat more efficient acceleration structure, they were able to achieve this by exploiting coherence and taking advantage of SSE SIMD instructions to trace packets of rays. Their packet-based ray tracer created groups of spatially coherent rays that were simultaneously traced through their kd-tree acceleration structure; each ray in the packet performed each traversal operation in lockstep. This packet-based approach enabled effective use of SIMD extensions on modern CPUs, increased the computational density of the code, and amortized the cost of memory accesses across multiple rays.

The next major performance advancement came in 2005, when Reshetov et al. presented their MLRT algorithm [67] for the kd-tree. One of the key advances they introduced was to extend the idea of packet traversal to traversing a frustum that bounds the rays in the packet. By doing this, the cost of a traversal step became independent of the number of rays bounded by the frustum. Provided the rays remained coherent and traversed the same kd-tree nodes, this allowed an arbitrary number of rays to be traced for the same cost as traversing a single frustum. For camera rays and shadow rays this is often the case and so for ray casted scenes with hard shadows they were able to achieve an order of magnitude improvement over previous ray tracers.

Wald et al. then applied the frustum traversal principles to grids [80] (see Chapter 5 for details) and BVHs [78] the following year. Although in the case of BVHs, rather than traverse a frustum, interval arithmetic was used and as soon as a single ray in the ray packet chose to enter a child node, ray-node testing would terminate and all the rays in the packet would enter that node. In this way, most of the traversal steps were independent of the number of rays in the packet, similar in effect to the pure frustum traversal methods used in kd-trees and grids.

2.3 Dynamic Scenes

From essentially the beginning of interactive ray tracing, ray tracers have provided some level of support for dynamic scenes. The interactive ray tracer that Parker et al. presented a decade ago allowed animated objects, but they had to exist outside of the acceleration structure, which meant that only a few objects could be animated [62].

Reinhard et al. [65] handled dynamic scenes through constant time insertions and deletions of individual moved objects from a grid acceleration structure. Although this allowed for fully dynamic scenes without rebuilding the entire acceleration structure from scratch, it introduced a performance penalty during ray traversal due to a less cache friendly grid structure. This also did not scale efficiently to scenes that contain triangles that are predominantly dynamic.

Lext et al. [53] and Wald et al. [77] both showed how certain simple types of animations could be handled in a ray tracer while still being accelerated. Their methods were limited mainly to rigid body animations that used hierarchical transformations, since this allowed them to avoid rebuilding the entire acceleration structure. Rather than modify all the objects in a subtree that had a transformation applied to it, the subtree stayed the same and instead transformed the rays that entered the subtree. In addition to only being limited to hierarchical transformations, another disadvantage was that the traversal became slightly more complex and slower as transformations were applied to the ray. Wald et al. also supported fully dynamic animations by rebuilding their kd-tree; however, due to the very slow rebuild time of their kd-tree, they were limited to small animations of up to a few thousand triangles.

For several years, interactive ray tracers were only able to handle a small subset of dynamic scenes. However, in the past couple years support for all types of dynamic scenes has emerged [82].

2.3.1 Grids

As Wald et al. [80] and later Lagae et al. [51] demonstrated, grids can be rebuilt from scratch extremely quickly. This makes grids suitable for handling fully dynamic scenes. Furthermore, Ize et al. have shown that grid rebuilds also parallelize very well and are able to scale to many cores as long as there is enough memory bandwidth available [41]. Fully dynamic scenes containing 10M triangles have been rendered interactively on multicore machines.

2.3.2 Bounding Volume Hierarchies

A BVH is defined through two parts: the tree topology, and each tree node's bounding volume. BVHs have the unique property that if objects in the BVH merely deform, then instead of rebuilding the complete BVH from scratch, the topology can be left unchanged and only the BVH nodes' bounding volumes need be *refit*. Although this refitted BVH may have a different and potentially less efficient tree structure than one built from scratch, the refitted BVH will nonetheless be correct. Refitting a BVH is extremely fast, and often less costly than the associated animation updates to the triangles. Since most animations can follow this requirement, BVHs are usually the acceleration structure of choice for most animations [52, 78, 82].

If full rebuilds are required, Wald has shown that the BVH can be rebuilt at close to interactive rates and can be further parallelized so that the rebuild can occur even faster [76]; however, the rebuild is still not as fast as a grid.

2.3.3 Binary Space Partitioning Trees

Aside from hierarchically transformed scenes, true BSPs are currently not suitable for fully dynamic scenes. Kd-trees, on the other hand, have been shown to handle fully dynamic scenes by completely rebuilding the kd-tree [69]. However, in order to accomplish this, an approximate lower quality SAH build is used, which results in rendering times that are about two-thirds as fast. Furthermore, interactive rebuilds are often achieved only after running a parallel build on four cores. For this reason, kd-trees are usually not used for dynamic scenes.

CHAPTER 3

RAY TRACING WITH THE REAL BSP

3.1 Introduction

Ray tracing with a BSP tree has traditionally meant using the axis aligned variant of the BSP tree—the kd-tree. This focus solely on axis-aligned planes is somewhat surprising, since in theory, a BSP should be far more robust to arbitrary geometry. In particular, it is relatively easy to "break" a kd-tree with non-axis aligned geometry: consider a long, skinny object that is rotated. While aligned to an axis, the kd-tree would be highly efficient, but when oriented diagonally, the kd-tree would be severely hampered. With a BSP, in contrast, rotating the object should not have any effect at all. In addition, since every kd-tree is expressible as a BSP tree, a properly built BSP tree could never perform worse than a kd-tree, and has the additional flexibility to place even better split planes that a kd-tree could not. Despite being theoretically superior on all counts, they are nevertheless not used in practice.

We believe that this discrepancy between theory and praxis stems from three widespread assumptions. First, it is believed that traversing a BSP tree is significantly more costly than traversing a kd-tree, since computing a ray's distance to an arbitrarily oriented plane requires a dot product and a division, whereas for a kd-tree it requires only a subtraction and a multiplication. Furthermore, a BSP tree node requires more storage than a kd-tree node. Thus, any potential gains through better oriented planes would be lost due to traversal cost. Second, the limited accuracy of floating point numbers is believed to make BSP trees numerically unstable, and thus useless for practical applications. Third, the (much!) greater flexibility in placing a split plane makes building (efficient) BSP trees much harder than building kd-trees. For a kd-tree, when splitting a node with n triangles there are only $6n$ plane locations where the number of triangles to the left and right of the plane changes; for a BSP tree, the added two dimensions for the orientation of the plane produces $O(n^3)$ possible planes. Instead, recursively placing planes through randomly picked triangles is likely to produce bad and/or degenerate trees, and would likely result in poor ray tracing performance. Thus, it is generally believed that naively building BSPs results in bad BSPs, whereas building good BSP trees is intractable.

In this chapter, we show that none of these assumptions is completely true, and that building a BSP-based ray tracer—when using the right algorithms for building and traversing—is indeed possible, and that it can be quite competitive to BVH or kd-tree-based ray tracers. While the numerical accuracy issues requires proper attention during both build and traversal, we will demonstrate our ray tracer on a wide variety of scenes, and with both ray casting and path tracing. We also show that the well-known surface area heuristic (SAH) easily generalizes to BSP trees, and that SAH optimized BSP trees—though significantly slower to build than kd-trees—can indeed be computed with tractable time and memory requirements. Based on a number of experiments, we show that our BSP-based ray tracer is at least as stable as a kd-tree-based one, that it is always at least roughly as fast as a highly optimized kd-tree, and that it can often outperform it.

3.2 Background

3.2.1 Binary Space Partitioning Trees

BSPs were first introduced to computer graphics by Fuchs et al. to perform hidden surface removal with the painter's algorithm [27]. As its name describes, a BSP is a binary tree that partitions space into two half-spaces according to a splitting plane. This property allows BSPs to be used as a binary search tree to locate objects embedded in that space. If a splitting plane intersects an object, the object must be put on both sides of the plane. This property can in theory lead to poor quality BSP trees with $\Omega(n^2)$ nodes for certain configurations of n nonintersecting triangles in $\Re^3$ [63]. However, in practice this will not occur and space should be closer to linear. In fact, if we only have *fat* triangles—which means there are no long and skinny triangles—then there exist BSP trees with linear size [19]. Assuming the tree is well-balanced, query time is usually $O(\log n)$. Build time depends on the algorithm used to pick the splitting planes. An algorithm that chose the optimal splitting planes would likely take at least exponential time, which is not feasible. For this reason, splitting planes are usually chosen at random, to divide space or the elements in half, or using some other heuristic such as the greedy SAH [30, 56].

Kammaje and Mora showed that using a restricted BSP with many split planes resulted in fewer node traversals and triangle intersections than their kd-tree, but for almost all their test scenes, the BSP resulted in slower rendering times [44]. One possible reason for this is that the traversal differs from a kd-tree traversal too much and ends up being too expensive. Another reason might be that at the lower levels of the tree, the restricted BSP still suffers from the same problem as the kd-tree, in that it is not able to cleanly split triangles apart. In a certain sense their data structure inherits the disadvantages of both kd-trees (not being able to perfectly adapt) and BSP trees (costly traversal). The RBSP build that Kammaje and Mora used was very slow, taking about 10 hours to build a tree for a million triangle model. Budge et al. [11] showed how the RBSP build could be made one to two orders of magnitude faster through the clever use of dynamic programming. Unfortunately, the dynamic programming technique they used required a fixed number of splitting planes and does not work on general BSPs.

Constrained BSPs were introduced for accelerating the manipulation of surfels used in point-based modeling [26]. A CBSP allows for arbitrary splitting planes as long as the number of faces of the resulting node is equal to or less than 6 (in 3D). Although this constraint does sometimes force the CBSP to use two splits when a BSP would require only one, for point-based modeling this constraint simplifies certain operations and is seen as advantageous. A CBSP could be useful for ray tracing if it allowed for faster builds, faster traversals, or better numerical precision. Unfortunately, it is not clear whether any of these benefits exist.

3.3 Building the BSP

The BSP build is conceptually very similar to the standard kd-tree build. The differences occur mainly in the implementation in order to handle the decreased numerical precision, computing the surface areas of a general polytope (as opposed to an axis-aligned box), and deciding which general split planes to use.

3.3.1 Intersection of Half-Spaces

The SAH requires the surface area of a node. For kd-trees this is trivial to calculate, but a node in a BSP is defined by the intersection of the half-spaces that make up that node. The intersection of half-spaces defines a polyhedron, and more specifically in the case of BSPs, a convex polytope since it can never be empty or unbounded (we place a bounding box over the entire object). We thus need to find the polytope in order to compute the surface area. As with Kammaje and Mora, we form the polytope by clipping the previous polytope with the splitting plane to get two new polytopes, and then simply sum the areas of the polygonal faces [44].

3.3.2 Handling Numerical Precision

We use the SAH, which requires us to compute the surface area of each node. As in [44], we find the surface area by computing the two new polytopes formed by cutting the parent node with the splitting plane. Care must be taken to make sure that the methods for computing the new polytope are robust and do not break down for very thin polytopes, which are quite common. Numerical imprecision makes it difficult to determine whether a vertex lies exactly on a plane. Assume we have a triangulated circle and a plane that is supposed to lie on that circle. If it is axis-aligned, for instance on the y-axis, determining whether each vertex lies on the plane is simple since all the vertices will have the same y value. But if we rotate the circle by 45 degrees so that the plane is now on $x = y$, the plane equation may not evaluate exactly to zero for these vertices. Consequently, we need to check for vertices that are within some epsilon of the plane. The distance of the vertices from the true plane is small; however, because we must use the planes determined from the triangles, the error can become much larger. If the triangles are very small and the circle they lie on very large, then a plane defined by a triangle on one side of the circle might end up being extremely far away from a triangle on the opposite side. If the epsilon is too small, tree quality will suffer since many planes will be required to bound the triangle faces when a single plane would have sufficed. This forces us to pick a much larger epsilon. However, if the epsilon is too large, then a node will not be able to accurately bound its triangles. This can cause rendering artifacts during traversal. For this reason we must also include this epsilon distance in the ray traversal.

3.3.3 BSP Surface Area Heuristic

We support a tree with a combination of general BSP and axis-aligned kd-tree nodes. Therefore, we have two node traversal costs, C_{BSP} and $C_{\text{kd-tree}}$, that we must use with the SAH. However, using these directly in the SAH results in BSP nodes being used predominantly over the kd-tree nodes even when C_{BSP} is set to be many times greater than $C_{\text{kd-tree}}$. The reason for this is the assumption that a split creates leaves with costs linear in the number of triangles. When there is only one constant traversal cost, this works well since the traversal cost affects only the decision of performing the split versus terminating and creating a leaf node. However, in our case we need to use the traversal cost not just to determine when to create a leaf, but also whether a BSP split, with its more expensive traversal, is worth using over a cheaper kd-tree traversal. Unfortunately, this linear intersection cost will quickly dwarf the constant traversal cost, causing the optimal split to be based almost entirely on which splitting plane results in fewer triangle intersection tests. We handle this problem by making C_{BSP} vary linearly with the number of primitives so that $C_{\text{BSP}} = \alpha C_i(n-1) + C_{\text{kd-tree}}$, where α is a user tunable parameter (we use 0.1). If after evaluating all splitting planes we find that the cost of splitting is not lower than the cost of creating a leaf node, then we evaluate all the BSP splitting planes again but this time with a fixed C_{BSP}. In practice this works much better than using a fixed cost; however, it is possible that an even better heuristic exists.

3.3.4 Which Planes to Test

At each node we must decide with which splitting planes to evaluate the SAH. Although a kd-tree has infinitely many axis-aligned splitting planes to choose from, Havran showed that only those splitting planes that were tangent to the clipped triangles (perfect splits) were actually required [31]. Since there are only 6 axis-aligned planes per triangle that fulfill this criteria, this allowed for $O(n)$ split candidates per node. Directly extending Havran's results to handling the additional degrees of freedom in choosing the normal would result in $O(n^3)$, or possibly even higher, split candidates, which is often impractical.

To maintain a practical build time, we limit ourselves to only $O(n)$ split candidates for each BSP node, at the expense of possibly missing some better splitting planes. For each triangle in a node, the split candidates we pick are the plane that defines the triangle face (auto-partition), the three planes that lie on the edges of the triangle and are orthogonal to the triangle face, and the same six axis-aligned splitting planes used by the kd-tree. Note that the first four splitting planes require a general BSP and would not work with a RBSP.

3.3.5 Build

Our build method is similar to those for building kd-trees with SAH, except that we are not able to achieve the $O(n \log n)$ builds that are possible for kd-trees [79] since we need to partition the triangles along an arbitrary direction. We use the same build algorithm as for the standard naive $O(n^2)$ kd-tree build [79]. At this point, we cannot use the faster lower complexity builds since we cannot sort the test planes. However, we are able to lower the complexity by using a helper data structure for counting the number of triangles on the left, right, and on the plane. This structure is a bounding sphere hierarchy over the triangles, where each node has a count of how many triangles it contains, so that if the node is completely to one side of the plane, the triangle count can be immediately returned. We build this structure using standard axis-aligned BVH building algorithms, so the tree can be computed extremely quickly [76] compared to the BSP build time. If all the triangles lie on the plane, then our current structure will end up traversing all the leaves and take linear time for that candidate plane. This worst case time complexity, however, will not increase the BSP build complexity since we would have had to spend linear time counting anyway. This can occur for portions of a scene, such as the triangles that tesselate a wall. However, most of the scene will not lie on the same plane and so for well-behaved scenes, this helper structure will give the location counts in sublinear time. Although we must update this structure after every split, and that takes $O(n)$, that cost is the same as the $O(n)$ splitting plane tests performed, so it does not increase the complexity. Thus, for almost any scene we are able to achieve a subquadratic build. Like in kd-trees, clipping triangles to the splitting plane results in a higher quality build [33].

3.4 Ray Tracing Using Generalized BSPs

3.4.1 Traversal

A ray is traversed through a kd-tree by intersecting the ray with the split plane giving a ray distance to the plane, which allows us to divide the ray into segments. The initial ray segment is computed by clipping the ray with the axis-aligned bounding box. A node is traversed if the ray segment overlaps the node. Since the two child nodes do not overlap, we can easily determine which node is closer to the ray origin and traverse that node first (provided it is overlapped by the ray segment), thereby allowing an early exit from the second node.

We use exactly the same traversal algorithm for the BSP tree except for two modifications. First, computing the distance to the plane now requires two dot products and a float division instead of just one subtraction and a multiplication with a precomputed inverse of the direction. Second, due to the limited precision of floating point numbers, the stored normal for any non-axis-aligned plane will always slightly deviate from the actual “correct” plane equation; thus, we cannot use the computed distance value directly, but have to assume that all distances within epsilon distance of the plane could reside on either side and so should traverse both nodes. Primarily because of the two dot products and the division, a BSP traversal test is significantly more costly; in our implementation, profiling shows the BSP traversal is roughly $1.75\times$ slower than the kd-tree traversal. We thus use a hybrid approach where axis-aligned splits can still use the faster kd-tree style traversal.

The initial ray segment is still computed by clipping the ray with the original axis-aligned bounding box, exactly as with kd-trees. Though we could in theory use an arbitrary or more complex bounding volume, such as in Kammaje and Mora's RBSP, we use the bounding box for reasons of simplicity. This has the added advantage that it provides a very fast rejection test for rays that completely miss the model.

3.4.2 Packet Traversal

Although we are primarily interested in single-ray traversal, as a proof of concept we have also implemented a SIMD packet traversal variant. Although for single-ray traversal the BSP and kd-tree traversals are nearly identical except for the distance computation, the packet traversal has a few significant differences. In particular, for kd-tree traversal one usually assumes that packets have the same direction signs, and splits packets with different signs into separate subpackets. Packets must have rays with matching direction signs so that the signs of the first ray can then be used to determine the traversal order of any given kd-tree node—since kd-tree planes are axis-aligned, once all rays in a packet have the same direction signs they will all have the same traversal order.

Since BSP planes can be arbitrarily oriented, this method of guaranteeing that a packet will always have the same traversal order no longer applies. Thus, traversal order must be handled explicitly in the traversal loop, including the potential case that different rays in a packet might disagree on the traversal order. For arbitrary packets, this would actually require us to be able to split packets during traversal. We avoid this by currently considering only packets with common origin, in which case one can again determine a unique traversal order per packet-based on which side of the plane the origin lies. This test is actually the same as the one proposed by Havran for single-ray traversal [31], except in packetized form. The resulting logic is slightly more complicated than the "same direction signs" variant, but the overhead is small compared to the more costly distance test, and it is independent of the orientation of the plane.

Since currently all our packets do have a common origin, the current implementation is sufficient for our purposes (the kd-tree code is optimized for common origins, too), but for practical applications this limitation would need to be addressed, perhaps by using Reshetov's omnidirectional kd-tree traversal algorithm [66].

3.4.3 Traversal-based Triangle Intersection

Since the BSP is able to perfectly bound most triangles and performs ray-plane intersection tests at each traversal, we are able to do ray-triangle intersection tests in the BSP for very little extra cost. This requires a small amount of extra bookkeeping during traversal, but by the time a ray reaches a leaf, the hit point can be found for free. We accomplish this by storing into each leaf node a flag for whether the node can use traversal-based intersection and the depth of the splitting plane that corresponded to the triangle face. These values can be stored into the node without using any extra space. During traversal we keep track of the current tree depth and the depth of the current near and far planes. With this, we are able to determine whether the triangle lies on the near or far plane, and if so, we get the hit point for free from the already computed ray-plane hit points used in the standard BSP/kd-tree traversal. This removes the need for most explicit ray-triangle intersection tests.

3.4.4 Data Structure

Optimized kd-trees generally use an 8 byte data structure where 2 bits specify the plane normal (essentially the axis), 1 bit for a flag specifying whether the node is internal or leaf, and 29 bits specify the index of the child nodes in the case of internal nodes, or the number of primitives in the case of leaf nodes. The remaining 4 bytes give the floating point normal offset. The BSP node is essentially the same, except that we must store 3 floats for the full plane normal. The data required by the traversal-based triangle intersection is only needed by the leaf nodes, so we can easily store that in the 12 bytes used the by normals in the internal nodes. BSP nodes therefore require 20 bytes per node. The total storage requirements can be computed by multiplying the number of nodes by 20 bytes.

3.5 Results

Measurements were taken on a dual 2 GHz clovertown (8 cores total). Build times are using a single core and render times using all 8 cores. We render both ray cast images using only primary rays with no shadows or other secondary rays, and simple brute force Kajiya style path traced images with multiple bounces off of lambertian surfaces. The ray casting lets us test very coherent packets, in contrast to the path tracing that exhibit extremely incoherent secondary ray packets. Using an optimized path tracer might result in faster render times, but it would not affect how the BSP compares with other acceleration structures.

We compare the BSP against an optimized kd-tree acceleration structure built using the $O(n \log^2 n)$ SAH with triangle clipping, and against the highly optimized SAH interval arithmetic (IA) BVH from [78]. Both the BSP and kd-tree can traverse rays in either single ray traversal or using SIMD packet traversal, but it cannot switch between the two during traversal. Unlike Reshetov's MLRTA [67], neither our BSP nor our kd-tree use frustum traversal or entry point traversal. Were they to do so, it is expected that the kd-tree would be faster than the BVH [78], and those techniques could also be applied to BSP traversal. The BVH always uses interval arithmetic traversal with SIMD packets, with the traversal performance often gracefully degenerating to single ray BVH performance for incoherent packets.

The kd-tree build can be modified so that if the chosen SAH split fails to satisfy certain conditions, such as it creates a leaf node with too many triangles, then a spatial median split is used instead in the hopes that it will allow the resulting child nodes to use splits that end up improving the overall SAH tree cost. This can often result in improved ray tracing times. In the case of a BVH, triangles can be split so that the BVH can produce higher quality SAH splits [18, 25]. The result in both the kd-tree and BVH cases is that less triangle intersections need be performed in exchange for more node traversals with the expectation that the final rendering time will improve. However, the kd-tree and BVH we use do not make use of these techniques.

We use a variety of scenes (Figures 3.1 and 3.2) for our tests, some of which were chosen because they highlight the BSP's strengths and others because they highlight the kd-tree's strengths. Section 9 of the UNC power plant and the UC Berkeley Soda Hall show how real scenes can contain many non-axis-aligned triangles for which the BSP is able to significantly outperform competing acceleration structures, in this case, by up to 3×.

The conference room was chosen as an example of a scene for which axis-aligned acceleration structures excel since many of the triangles are axis-aligned. As expected, the BSP does not show a strong advantage over other structures; however, it still proves to be slightly faster than the kd-tree since most of the tree ends up using kd-tree nodes and near the leaves, the BSP is able to further split triangles whereas the kd-tree and BVH would normally have stopped and created leaves containing multiple triangles.

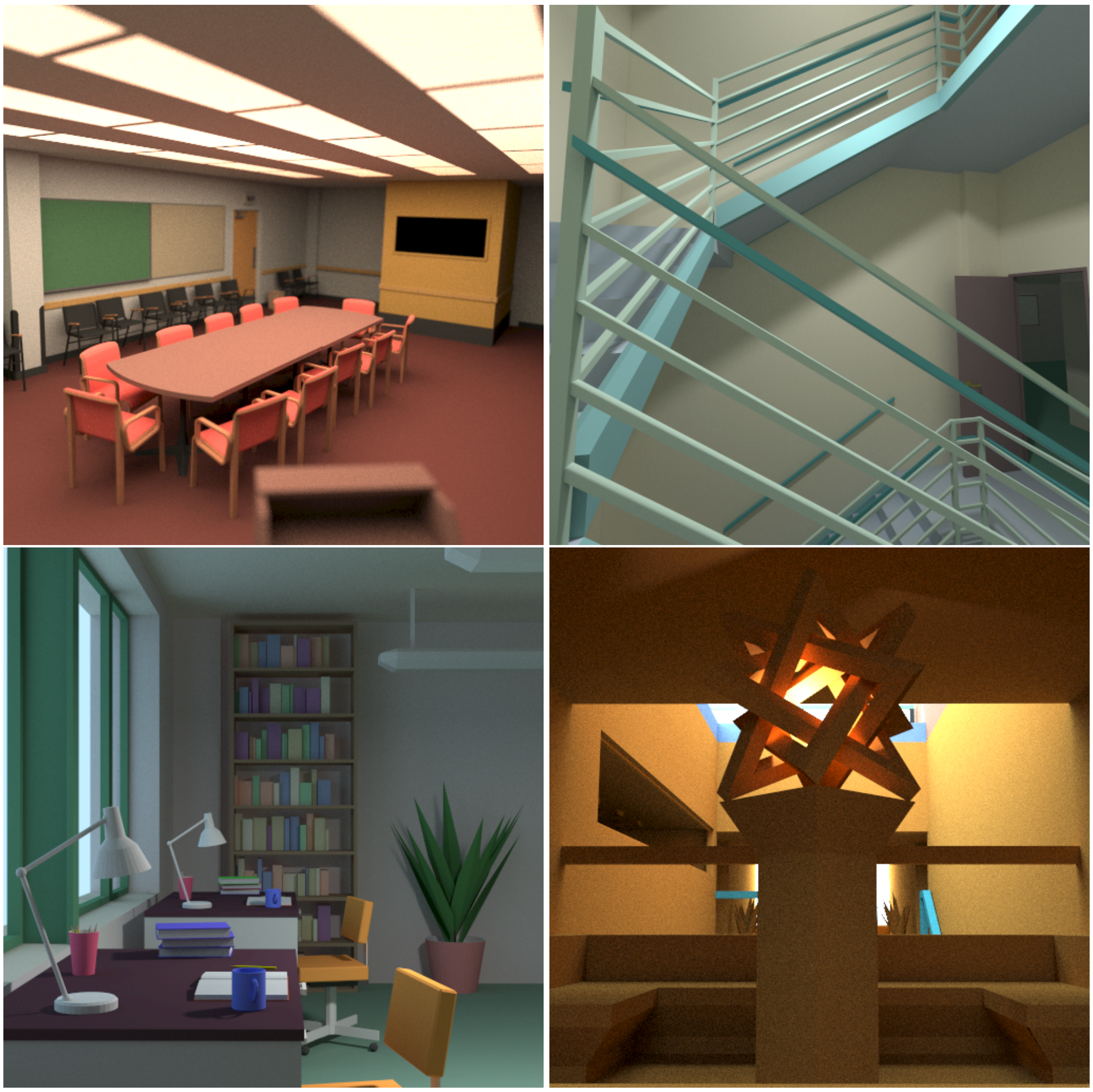


Figure 3.1. Path traced architectural scenes. The 283 thousand triangle conference room and three views of the 2.2 million triangle UC Berkeley Soda Hall model, rendered with a path tracer. For these views, our BSP outperforms a kd-tree by 1.1×, 1.3×, 1.2×, and 2.5×, respectively (and by 1.1×, 1.8×, 1.2×, and 2.4×, respectively, for ray casting)

The Stanford bunny is a traditional example of a character mesh. It is a well-behaved scene that does not favor any particular acceleration structure since most triangles are of similar size, not axis-aligned, and not long and skinny. Note that the BVH does better than the BSP and kd-tree at path tracing the bunny because the average ray packet remains coherent. This occurs since most secondary rays will immediately hit the background and terminate and so never have a chance to become incoherent. For enclosed (indoor) scenes,

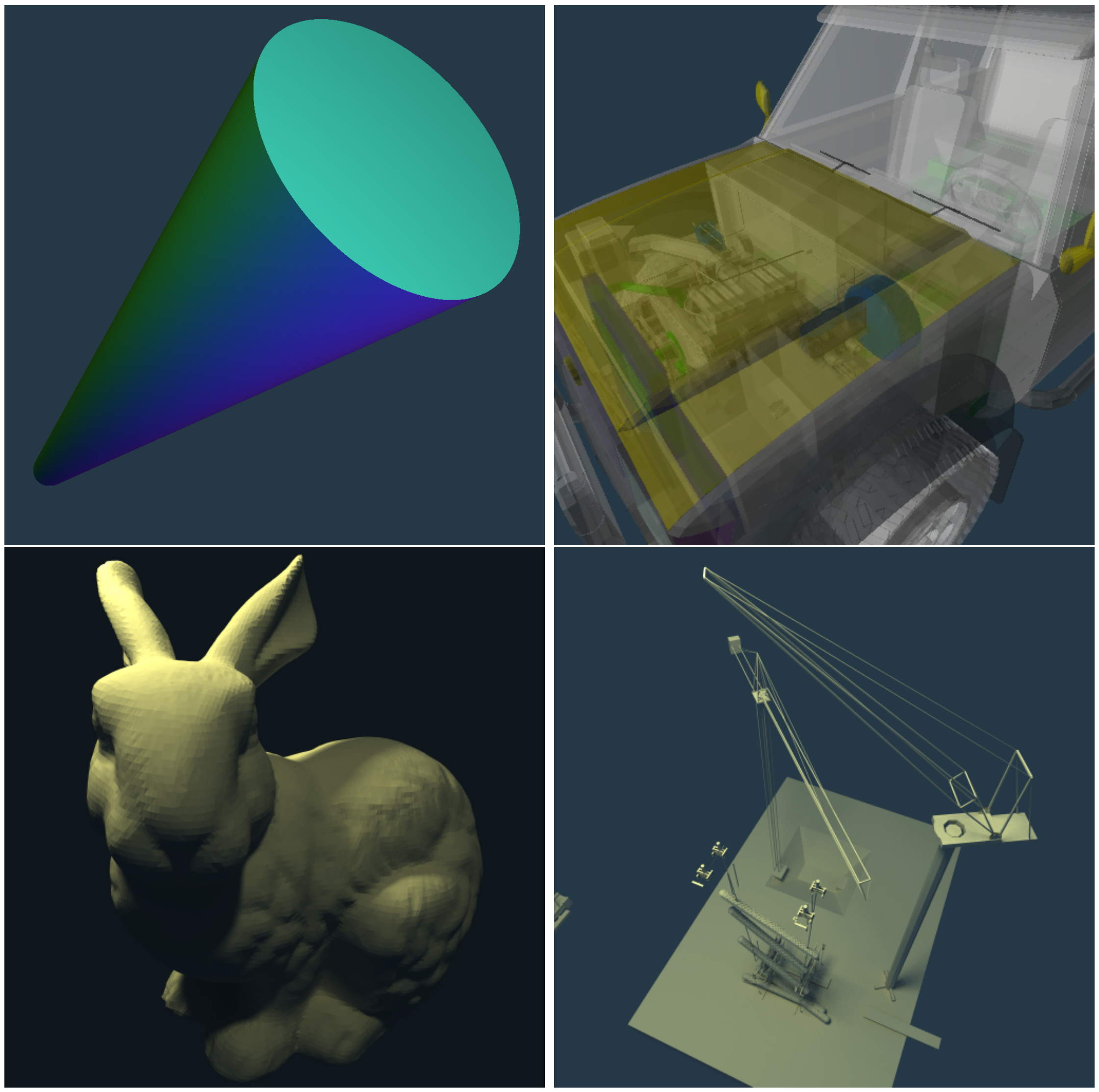


Figure 3.2. Ray cast and path traced scenes. Ray cast cylinder (596 triangles) and transparent pickup truck (183k), rendered 26× and 1.3× faster than the single ray kd-tree. Path traced bunny (69k), and section 9 of the UNC powerplant (122k) outperform the kd-tree by 1.2× and 2.4×.

rays can bounce around multiple times and therefore impact the overall time more than the primary rays.

The pickup truck used for bullet ray vision by Butler and Stephens [12] is a good example for where the expensive BSP build cost is computed once in a preprocess and then easily offset by having a faster interactive visualization. Although the bullet vision paper used a more complicated material to determine the amount of transparency, we use a fixed value for the transparency for these tests. However this does not affect the overall results when comparing acceleration structures.

As a stress case for other axis-aligned structures, and to showcase the abilities of the BSP, we compare against a non-axis-aligned cylinder tessellated with many long skinny triangles. Although this is clearly not realistic and strongly favors the BSP, it does help to illustrate the strengths the BSP has over other acceleration structures, and helps to explain why the BSP can outperform other acceleration structures.

Impressively enough in all the scenes with secondary rays, with the exception of the bunny, the single ray BSP is able to outperform all other acceleration structures including the SIMD packetized IA BVH acceleration structure. In the bunny scene the BSP is able to outperform only the kd-tree. The single ray BSP is always faster than the single ray kd-tree and the SIMD BSP is usually as fast or faster than the SIMD kd-tree.

The ray packet improvements for the BSP over the kd-tree are not as pronounced as the single ray improvements because ray packets benefit less from having triangles completely subdivided into separate nodes. This is because the ray packet will often be larger than the individual triangles.

In order to measure how including kd-tree nodes in the BSP improves performance, we also compare our BSP with a BSP that uses only general splitting planes and is built using a standard SAH; we refer to this as BSP-orig. BSP-orig is still novel since we are not aware of any general BSPs that are built with the SAH, and it still performs better than the kd-tree for some scenes. However, we do not advocate using it since the optimized BSP we present in this paper is almost always faster, easy to implement, and the trees are not much larger in size.

3.5.1 Statistical Comparison

From Table 3.1 we see that the BSP uses roughly as many traversal steps as the kd-tree, with a fraction of those traversals going through the more expensive BSP nodes. For this reason, the BSP will usually take roughly as much to slightly more time traversing than the kd-tree. However, the advantage of the BSP is that it performs many times less triangle intersections. Unfortunately, for scenes that a kd-tree does well on, such as the conference room, the possible speedup that the BSP can achieve over the kd-tree is limited since the kd-tree only spends about 30% of the time intersecting triangles and roughly 55% of the time traversing nodes. Amdahl's law thus states that even if we eliminated the intersection cost completely without slowing down the traversal, the overall speedup would only be 1.4×. We reduced the number of triangle intersections by 4×, which would have resulted in a 1.3× speedup if the traversal cost stayed the same. However, instead, the traversal cost also went slightly up so in the end we got a 1.1× speedup. This means the BSP cannot be significantly faster than the kd-tree unless it is also able to do many fewer traversals or the cost of a BSP traversal went down. Since our BSP is already very close to the minimum number of triangle intersections (even fewer with traversal-based intersection), a higher quality build, for instance from testing $O(n^3)$ possible splitting planes at each node, would only be able to significantly improve performance by reducing the number of traversals. However, for certain scenes or viewpoints, for instance the soda hall art or section 9 of the power plant, the amount of time the kd-tree spends on triangle intersections becomes quite high (two orders of magnitude in these examples) and for these situations, the BSP can offer very significant improvements; this is where the BSP is truly superior.

Increasing C_i will result in more aggressive splits, which ends up further reducing the number of triangle intersections, at the expense of more node traversals. When not performing traversal-based intersections, this causes a slight negative performance hit. Overall, the traversal-based intersection performs better since more triangles end up being tightly bound by splitting planes. The results presented in this paper are all using the lower intersection cost to build the BSP tree.

Table 3.1 shows that the BSP-orig is able to perform fewer triangle tests than the optimized BSP and often performs less total traversal steps. However, since all those traversal steps are done using the more expensive BSP traversals, the actual render time ends up increasing as seen in Table 3.2.

Table 3.3 shows that the BSP is able to subdivide most triangles into individual leaves. Contrast this with the kd-tree build for the conference room, which has 13 leaves with more than 100 triangles and about 100K nodes with more than four triangles, and roughly as many nodes with one triangle as with two or three triangles. Another interesting point is that whereas the majority of nodes in the BSP tree use general BSP nodes, Table 3.1 shows that most of the nodes actually visited during traversal use the kd-tree style node. This is explained by noting that most kd-tree nodes are higher up in the tree where they are more likely to be visited, whereas the BSP nodes are usually found near the leafs.

Table 3.1. Per ray statistics. The total number of ray-triangle tests is the sum of the expensive standard tests and the very cheap traversal-based tests. The total number of node traversals is also the sum of the BSP and kd-tree node traversals. BSP-orig is a BSP that does not use kd-tree traversals. If traversal-based triangle intersection tests are not used in the BSP, then the number of standard triangle tests is roughly the sum of the standard tests and the traversal-based tests mentioned in the table.

	Ray-Triangle Tests		Node Traversals	
Ray Casted	standard	traversal-based	BSP	kd-tree
bunny				
BSP-orig	0.07	0.40	21.6	-
BSP	0.11	0.36	5.8	18.4
kd-tree	2.3	-	-	27.8
conference				
BSP-orig	0.77	0.52	32.8	-
BSP	0.82	0.52	3.2	31.8
kd-tree	3.5	-	-	33.5
power plant 9				
BSP-orig	0.04	0.24	23.3	-
BSP	0.09	0.21	4.9	19.9
kd-tree	18.5	-	-	25.7
cylinder				
BSP-orig	0.07	0.33	10.9	-
BSP	0.08	0.34	8.6	3.1
kd-tree	182.4	-	-	15.2
truck				
BSP-orig	1.4	0.40	27.4	-
BSP	1.4	0.45	7.1	22.3
kd-tree	7.5	-	-	26.5
sodahall stairs				
BSP-orig	0.12	0.95	48.1	-
BSP	0.18	0.91	8.5	42.8
kd-tree	13.7	-	-	58.6
sodahall room				
BSP-orig	0.26	0.86	75.8	-
BSP	0.40	0.81	7.7	51.2
kd-tree	5.4	-	-	61.9
sodahall art				
BSP-orig	0.17	0.89	52.3	-
BSP	0.14	0.94	4.0	42.9
kd-tree	20.6	-	-	41.8

Table 3.1. continued.

	Ray-Triangle Tests		Node Traversals	
Path Traced	standard	traversal-based	BSP	kd-tree
bunny				
BSP-orig	0.26	0.28	30.2	-
BSP	0.32	0.26	8.3	25.1
kd-tree	4.6	-	-	36.8
conference				
BSP-orig	0.87	0.35	29.0	-
BSP	0.98	0.35	2.9	25.5
kd-tree	4.2	-	-	29.2
power plant 9				
BSP-orig	0.11	0.18	24.4	-
BSP	0.12	0.17	5.1	20.8
kd-tree	17.5	-	-	26.6
sodahall stairs				
BSP-orig	0.24	0.57	39.2	-
BSP	0.27	0.57	4.1	33.6
kd-tree	4.9	-	-	38.5
sodahall room				
BSP-orig	0.32	0.51	46.9	-
BSP	0.39	0.48	4.3	36.4
kd-tree	5.9	-	-	41.4
sodahall art				
BSP-orig	0.22	0.70	47.4	-
BSP	0.31	0.69	8.6	43.2
kd-tree	44.4	-	-	37.2

Table 3.2. Rendering performance. Frame rate for rendering the 1024spp 512x512 path traced scenes and the 1024x1024 ray cast scenes. BSP-orig is the BSP without the kd-tree node optimization and BSP-tri is the BSP with traversal-based intersection.

	Single Ray				SIMD Ray Packet		
	BSP-orig	BSP	BSP-tri	kd-tree	BSP	kd-tree	BVH
Path Traced (frames per minute)							
bunny	0.874	0.954	0.963	0.796	-	-	1.306
conf	0.296	0.347	0.358	0.333	-	-	0.218
soda room	0.233	0.349	0.363	0.314	-	-	0.171
soda art	0.217	0.236	0.250	0.102	-	-	0.116
soda stairs	0.170	0.224	0.242	0.188	-	-	0.084
pplant9	1.51	1.74	1.76	0.731	-	-	1.111
Ray Traced (frames per second)							
cylinder	22.0	21.9	23.0	0.868	49.2	5.73	5.28
bunny	12.2	13.3	13.7	11.2	20.2	22.6	38.0
truck	2.46	2.75	2.80	2.19	-	-	1.87
conf	8.87	10.9	11.6	10.3	26.2	25.7	31.9
soda room	4.17	7.43	7.6	6.33	23.7	21.9	25.0
soda art	6.09	9.06	9.7	4.01	30.6	16.9	26.0
soda stairs	6.47	8.07	8.7	4.79	25.8	17.3	11.6
pplant9	13.3	16.1	16.4	4.95	33.8	19.2	17.8

3.5.2 Absolute Performance Comparison

The BSP should always be roughly as fast or faster than the kd-tree since the BSP can use kd-tree style traversal. Compared to the IA BVH, the BSP tends to be faster only for incoherent rays, as evident in the path traced benchmarks, and for scenes with many non-axis-aligned triangles. Performing the triangle intersection as part of the BSP traversal will result in performance improvements if most rays are intersecting triangles (indoor scenes or closeups of polygonal characters).

In the path traced bunny, many primary rays miss the bunny, and of those that do hit the bunny, most cast new rays that hit the background. As such, the ray packets are much more coherent than in interior path traced scenes, such as in the conference scene, and so the IA BVH is able to outperform the BSP.

In Figure 3.3 we compare the per pixel rendering time when performing just ray casting. In the first two columns, each pixel intensity corresponds to the amount of time taken to render that pixel when using a kd-tree and a BSP. The third column is the normalized difference of the first two columns, with white corresponding to the kd-tree taking longer and red with the BSP. In the power plant, soda hall, and cylinder scenes,

Table 3.3. Build statistics.

	cylinder	pplant9	conf	bunny	truck	sodahall
	596	122K	283K	69K	183K	2.14M
	BSP / kd	BSP / kd	BSP / kd	BSP / kd	BSP / kd	BSP / kd
build time	9.2s / 0.24s	36m / 29.2s	112m / 1.2m	19m / 14.3s	65m / 46s	23.6h / 6.3m
max tree depth	26 / 33	48 / 75	47 / 89	33 / 59	44 / 135	60 / 108
# nodes	15K / 13K	3386K / 1645K	9246K / 3921K	1470K / 988K	3363K / 2870K	28.9M / 27.9M
% kd-tree nodes	15% / 100%	37% / 100%	21% / 100%	27% / 100%	27% / 100%	31% / 100%
leaf (0 tris)	3.3K / 3.7K	734K / 181K	2128K / 746K	423K / 250K	774K / 376K	6.73M / 3.36M
leaf (1 tri)	3.9K / 412	783K / 171K	2196K / 279K	309K / 24K	713K / 277K	6.41M / 2.29M
leaf (2 tris)	142 / 333	125K / 264K	275K / 383K	3.3K / 158K	176K / 440K	1.13M / 5.29M
leaf (3 tris)	30 / 1.2K	45K / 130K	21K / 333K	107 / 51K	15K / 217K	130K / 2.39M
leaf (4 tris)	0 / 131	3.5K / 41K	3.1K / 108K	2 / 9.6K	3.0K / 73K	20K / 403K
leaf (> 4 tris)	0 / 603	3.7K / 36K	493 / 111K	0 / 905	618 / 52K	8.7K / 209K
max tris in leaf	3 / 301	17 / 64	8 / 101	4 / 7	20 / 119	600 / 600

the BSP clearly performs much better than the kd-tree due to the kd-tree not being able to handle the long skinny non-axis-aligned triangles. The brightened silhouettes on the bunny comparison shows how the BSP is able to quickly get a tight bound over the mesh. The BSP shows much less variation in render time compared to the kd-tree, which has substantial variation with some regions of the scene clearly showing "hot spots" where the kd-tree breaks down.

If we position the camera on one of these hot spots (white pixels), the BSP will have a much bigger advantage. For instance, even though for the camera view we used in the conference room the BSP is only slightly faster than the kd-tree, the BSP can get an order of magnitude improvement over the kd-tree simply by moving the camera to look at just the chair arms. The same principle applies to all the other scenes.

3.5.3 Build Time and Build Efficiency

The complexity of our BSP build algorithm for well-behaved scenes is subquadratic. From experimental observation, it appears to be around $O(n \log^2 n)$, which makes the complexity fairly close to that of kd-trees that can currently be built in $O(n \log n)$. However, the BSP is still almost 2 orders of magnitude slower. This is mainly due to the large cost in calculating the surface area of a node. Since the focus of this paper is not on fast builds, the BSP builder is left unoptimized.

These build times are clearly not interactive and are only useful as a preprocess. However, note that the full quality kd-tree build is also still not interactive. The $O(n \log n)$ kd-tree build from Wald and Havran [79] is 2–3$\times$ faster than our $O(n \log^2 n)$ kd-tree build, but also still a couple orders of magnitude too slow for interactive builds. In order to get interactive kd-tree and BVH builds, the tree quality must suffer. For coherent packet tracing, tree quality near the leaves is not as important since the packets are often as large as each node, and so using a lower quality approximate build will often result in only a slight increase in intersection tests. However, for incoherent packets where each ray intersects different primitives, the faster approximate builds will result in a very noticeable performance penalty as rays perform many more unneeded intersection tests.

3.6 Summary and Discussion

We have shown that a general BSP is as good or better than a kd-tree if build time can be ignored. For scenes with incoherent rays or with many long non-axis-aligned triangles, the single ray BSP is even able to outperform the packetized SIMD IA-BVH.

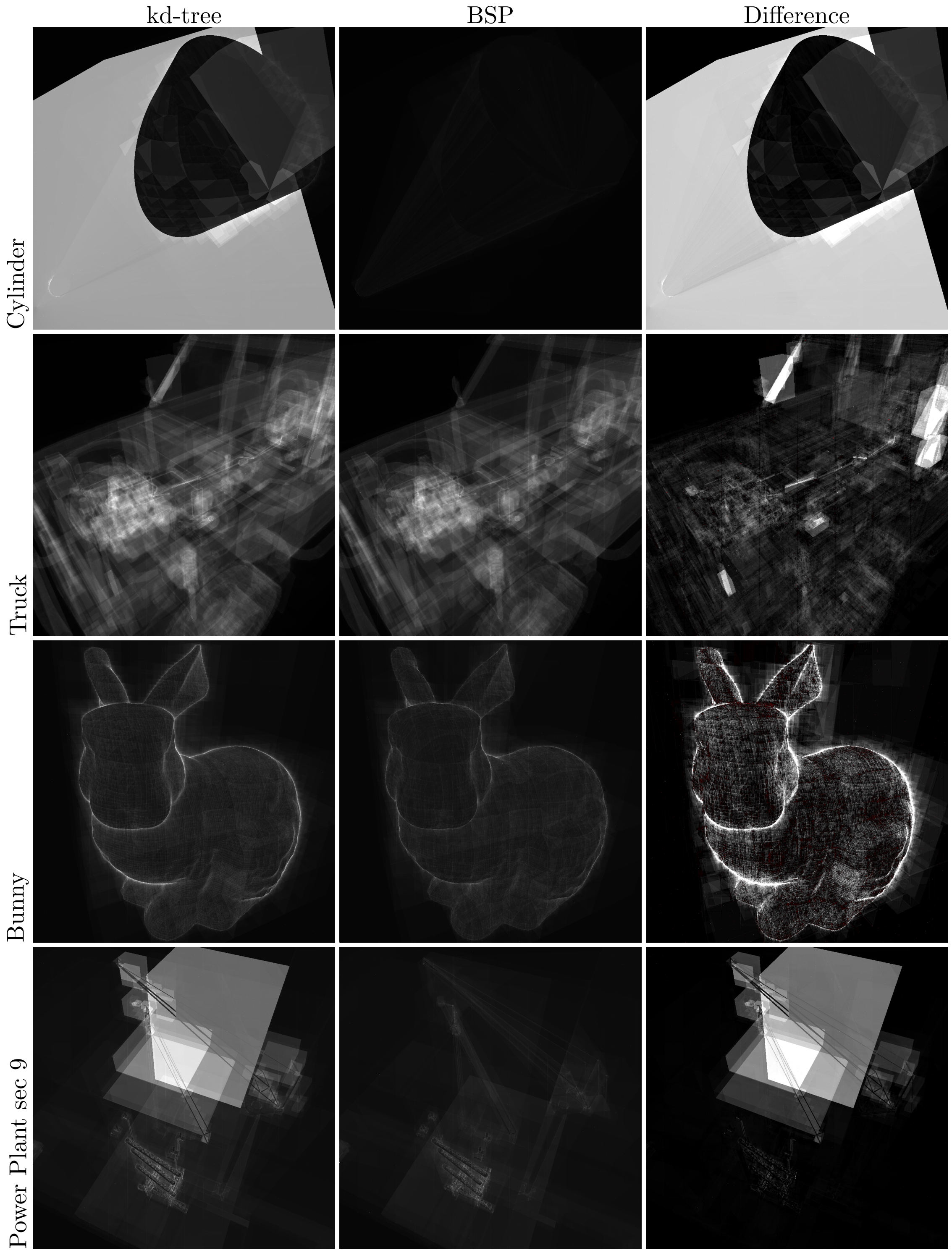


Figure 3.3. Rendering time visualization. Rendering time difference (right) of kd-tree (left) with BSP (middle). The time visualizations are rendered so that increasing intensity corresponds to increased rendering time for the pixel. The time differences are the normalized differences of the BSP and kd-tree timings, with white corresponding to the kd-tree being slower and red with the BSP being slower.

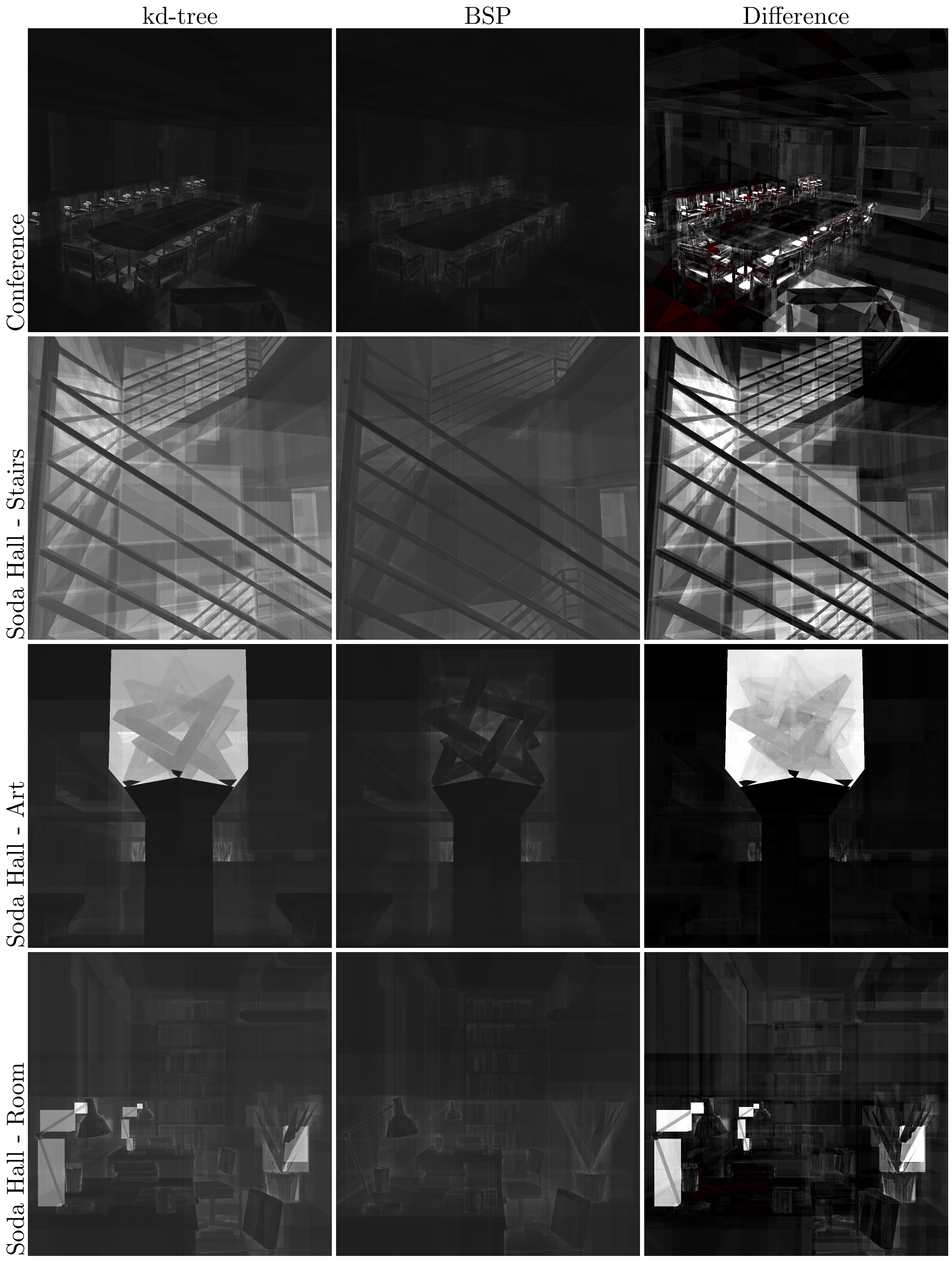


Figure 3.3. continued

Furthermore, the BSP is able to handle difficult scenes on which other axis-aligned-based acceleration structures break down. Even though the complexities of kd-trees and BSPs are roughly the same, in practice, the BSP tree allows for fewer triangle-ray intersection tests with only a minimal increase in the traversal cost. The complexity of our current SAH build is subquadratic, like the kd-tree, but it has a rather large constant term due to the expensive computation of the surface area formed by the intersection of half-spaces, which leads to slow build performance. However, there are still many ways to improve the build performance and we expect the BSP build to easily become much faster than it is currently. Furthermore, as interactive ray tracing begins to focus more on incoherent secondary rays, the fast approximate interactive builds will not perform as well, and so for static scenes, slower but full quality builds will once again be required. If the tree will be built offline as a preprocess, then spending an hour instead of a minute in exchange for more consistent frame rates with no trouble hot spots can be a very worthwhile trade-off for certain applications. Clearly, the BSP is currently not suitable in situations where fast build times are required.

Numerical precision issues cause problems in the build. During rendering, these precision issues can be easily handled by using an epsilon test when doing a standard check for whether a ray-plane intersection occurs on the near or far plane. This adds a negligible two additional addition instructions and occasionally results in a few extra unneeded traversals, but rendering quality is not affected.

In all of our tests, the BSP tree was always the fastest option for single ray traversal and is robust to scene geometry, unlike other acceleration structures that might perform well in some conditions but poorly in others. Looking at the time visualizations from Figure 3.3 we saw that many scenes exhibit hot spots when rendered with the kd-tree, but not with the BSP. If the user were to by chance zoom into one of these hot spots, the frame rate could go down by an order of magnitude as in the soda hall, cylinder, and conference room chair arm examples. In some applications this is unacceptable and the expensive BSP build would be fully justified, especially when used as a one time preprocess.

3.6.1 Future Work

Our implementation makes use of ray packets but not the frustum/IA-based traversal present in the fastest acceleration structures. Although this might be fine when there is low ray coherence, such as in path tracing, for very coherent rays, this can sometimes make up to a two to threefold difference in performance [67, 78, 80]. Extending Reshetov's MLRTA from kd-trees to BSPs would likely be the best method. Another optimization would be to dynamically use the packetized traversal for coherent packets and single ray traversal for incoherent packets. This would allow the BSP (and the kd-tree) to traverse the primary rays much faster. This would likely remove the advantage the BVH had in some scenes that were predominately primary rays, such as the path traced bunny.

We would also like to improve the build performance. This can be done in many ways. Optimizing the surface area calculation and parallelizing the build are two obvious options and would likely result in substantial speedups over our unoptimized code. Randomly testing only a subset of the possible splitting planes would trade build quality for build speed. Handling axis-aligned splits with a traditional kd-tree build should give a significant improvement, especially since many of the kd-tree nodes are near the top of the tree where nodes have more splitting plane candidates and so are more expensive. Since the top of the tree consists mainly of kd-tree nodes, we could also only use a kd-tree build for the top of the tree and switch over to general splits when there are less than a certain number of triangles in a node or if performing a kd-tree split results in a small improvement to the SAH cost. Another similar option is to use the RBSP build method of Budge et al. [11] in areas of the tree where quantized splitting plane normals do not negatively affect tree quality; this is likely a common condition for the top levels of the tree. This would result in one to two orders of magnitude speedup in the top of the tree and a possible overall speedup of 2–4$\times$, while resulting in roughly equal quality as a true BSP build.

Improving the selection of splitting planes could allow for further improvements in tree quality. For instance, when using a triangle edge as the split location, we set the plane normal to be orthogonal to the edge and triangle normal. This is an arbitrary decision and will not always result in an optimal split. For instance, if two triangles share an edge and form an acute angle, the splitting plane on that edge will place both triangles on the same side, which might not be desirable. Likewise, if all the triangles are axis-aligned, but form diagonal lines through stair-stepping, then the optimal split will be those that are tangent to as many triangles as possible.

Our heuristic for combing C_{BSP} and $C_{\text{kd-tree}}$ is an improvement over the standard SAH, but it is likely that an even better heuristic exists that would further result in performance improvements.

Extending this to handle more expensive primitives, such as spline patches, could result in substantial improvements since minimizing the number of ray-patch intersection tests would make a noticeable difference. The only required modification would be in choosing the candidate splitting planes and in removing the traversal-based triangle intersection. One possible way to select the planes would be to use the axis-aligned planes that make up the bounding box as well as randomly selecting planes that are tangent to the patch.

CHAPTER 4

GRID CREATION STRATEGIES FOR EFFICIENT RAY TRACING

4.1 Introduction

Acceleration structures are used in ray tracing to improve the time needed to compute ray-scene intersections at the expense of increased preprocessing time [5]. Popular methods include hierarchical spatial subdivision (e.g., octrees and kd-trees), hierarchical object subdivision (e.g., bounding volume hierarchies), and uniform spatial subdivision often called "grids." All of these methods can be mixed in the form of "meta-hierarchies" such as bounding volume hierarchies with grids or kd-trees in their leaves [48]. There is currently no agreement on what acceleration structure is best [82], and it is likely that the answer varies depending upon the details of the model.

In this chapter we examine some properties of the grid acceleration structure [16, 28], as well the parameters that should be used when using nested grid structures [43, 50]. We examine different build strategies that are implied by whether a model is manifold-like, and whether a model is made mainly of long-skinny triangles. Our approach is to use theory with simplifying assumptions, and to then experimentally test whether these assumptions prevent the theoretical results from applying in practice. We take our inspiration from the fact that an analysis on uniformly random rays and infinitesimal points in 3D (Figure 4.1, left) yields useful grid construction guidelines despite our straightforward analysis. We extend such simple analysis to long-skinny primitives that we approximate with lines and to models whose primitives lie along surfaces (Figure 4.1).

The guidelines for grid construction we suggest are the following:

- for scattered data composed of N compact primitives, a single-level grid with $O(N)$ grid cells has an asymptotically optimal tracing time of $O(N^{1/3})$;
- for scattered data composed of N long-skinny primitives, a single-level grid with $O(N^{3/2})$ grid cells has an asymptotically optimal trace time of $O(N^{1/2})$;

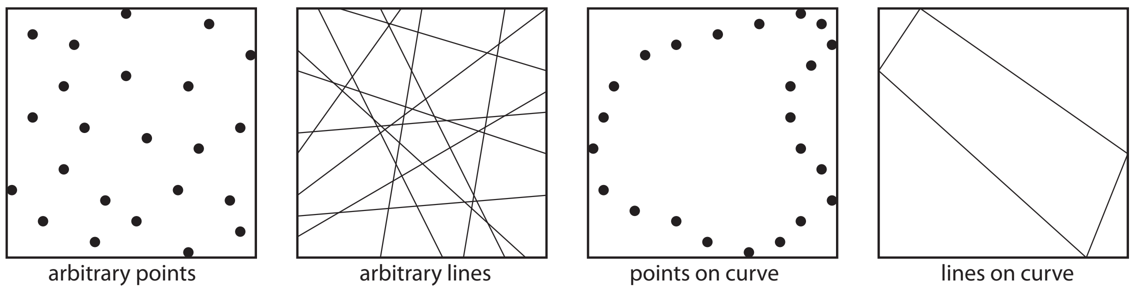


Figure 4.1. 2D versions of the scene types analyzed.

- for manifold-like models composed of N compact primitives, a two-level grid can reduce the time complexity of tracing to $O(N^{1/5})$ if $O(N^{3/5})$ grid cells are used for the top level;
- for manifold-like models composed of N compact primitives and d levels, $O(N^{3/(2d+1)})$ grid cells should be used for the top level, resulting in tracing asymptotic complexity of $O(N^{1/(2d+1)})$;
- for manifold-like models composed of long-skinny triangles, a two level grid should use $O(N)$ cells in the first level grid and results in a tracing complexity of $O(N^{1/3})$.

We believe that all but the first two observations are new. Cleary and Wyvill have a more complicated analysis for the first two observations that makes fewer simplifying assumptions [16].

Previously, recursive grid dimensions were chosen such that each level subgrid would always create $O(N)$ cells, where N is the number of primitives in that specific subgrid [43]. This results in only $O(N^{1/3})$ time complexity, instead of the $O(N^{1/(2d+1)})$ presented here. We were not able to discover any preexisting theory that helped guide in the choosing of each parameter value, which made choosing the correct parameters challenging, especially since they could vary substantially with N and the subgrid level. For instance, a two level grid would often see an order of magnitude difference in parameter values between the two levels. Our strategy not only can give better performance, it is also much easier to use since only one parameter needs to be adjusted and this parameter does not vary with the scene.

4.2 Classic Results for One-Level Grids

In this section we review the derivation of choosing the optimal grid resolution for simplified conditions. Details of this derivation can be found in classic works [16, 21, 43].

We then discuss how well the results of this simple analysis apply to several geometric models.

4.2.1 Theoretical Analysis

The terms and symbols we use are summarized in the Notation and Symbols page. We now summarize the classic analysis for one-level grids. We begin with several simplifying assumptions:

- the N primitives are all points (infinitely small);
- all rays hit the root box from the outside, and all are equally likely;
- the root box is a cube;
- each atomic function time can be treated as a constant;
- mathematical operations (as opposed to memory effects, etc.) dominate performance;
- the grid has $m^3 = M$ cells, and M can be treated as a continuous number so discrete math idiosyncrasies can be avoided.

Geometric probability (or the closely related integral geometry) shows that exactly m cells on average are hit by a ray. Regardless of the distribution of the points, an average of N/m^3 points occupy a single cell, so a ray tests (and always misses) an average of N/m^2 infinitesimal points on its way through the grid. The average time T for a single ray to compute the fact that it misses all points is:

$$T = T_{\text{setup}} + mT_{\text{step}} + \frac{N}{m^2}T_{\text{intersection}} \tag{4.1}$$

By treating m as a continuous number we can take a derivative and minimize T:

$$\frac{dT}{dm} = T_{\text{step}} - \frac{2N}{m^3}T_{\text{intersection}} \tag{4.2}$$

Which when set to zero implies that T is minimized when:

$$m = \sqrt[3]{N\frac{2T_{\text{intersection}}}{T_{\text{step}}}} \tag{4.3}$$

4.2.2 Empirical Test of Analysis

The simplicity of these assumptions is cause for concern, but unfortunately, the analysis is much more involved when these assumptions are relaxed [16, 21, 43, 85]. However, Equation 4.3 does seem to apply to at least some real models, and that happy fact is the inspiration for the simplified analyses we make later in this chapter. We deviate from the results given by our theory by using grid cells that are as close to a cube as possible (to allow for noncubical grid shapes), which in turn leads to the grid containing different numbers of divisions per dimension, as explained in [80]. Since m no longer represents the number of divisions, we use the computed value of M instead of m. The resulting value for M (from cubing the m in Equation 4.3) is:

$$M = N\frac{2T_{\text{intersection}}}{T_{\text{step}}} \tag{4.4}$$

The models in particular where the assumptions underlying the simple analysis are not too damaging are scanned models composed of relatively small triangles. In Figure 4.2 we have fit runtimes on several models to Equation 4.1 and have found the empirical behavior to be quite close to the theory with the caveat that the coefficients are slightly different for each model. The key ratio for optimization of runtime, $T_{\text{intersection}}/T_{\text{step}}$, is on average 4 for the models we have tested. This ratio and Equation 4.4 imply $M = 8N$. The relative performance of this M and the empirically determined optimal M for several models is shown in Table 4.1. As can be seen, for such scanned models, we only sacrifice a few percentage points in runtime by using our theoretical M for every model. Even the 282K triangle conference room that is even further from our assumptions closely follows our theory.

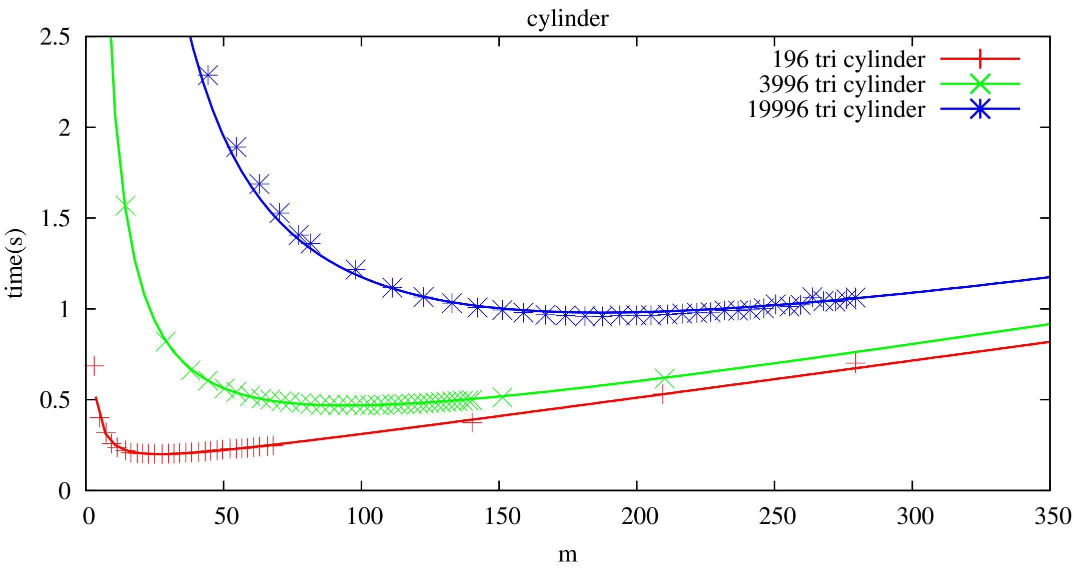
4.3 Long-Skinny Triangles and One-Level Grids

If the model is made up of long-skinny objects (as they would be in a tesselated cylinder), then each object will be added to more than $O(1)$ cells so the basic assumptions of the previous section do not apply. In practice, long-skinny objects get cut into more and more effective objects as more grid cells are added, so the benefits of grids are diminished. Rather than using points to approximate small objects as in the last section, we use line segments to approximate long thin objects.

4.3.1 Theoretical Analysis

In the same spirit of simplified analysis, we can assume each long-skinny object is actually a line segment spanning the entire model. This means as m increases, so does the total number of effective primitives in the model. Each line will intersect $O(m)$ cells. If all lines are equally likely, each will intersect an average of m cells. This changes Equation 4.1 to have an extra factor of m in the numerator of the last term:

$$T = T_{\text{setup}} + mT_{\text{step}} + \frac{N}{m}T_{\text{intersection}} \tag{4.5}$$



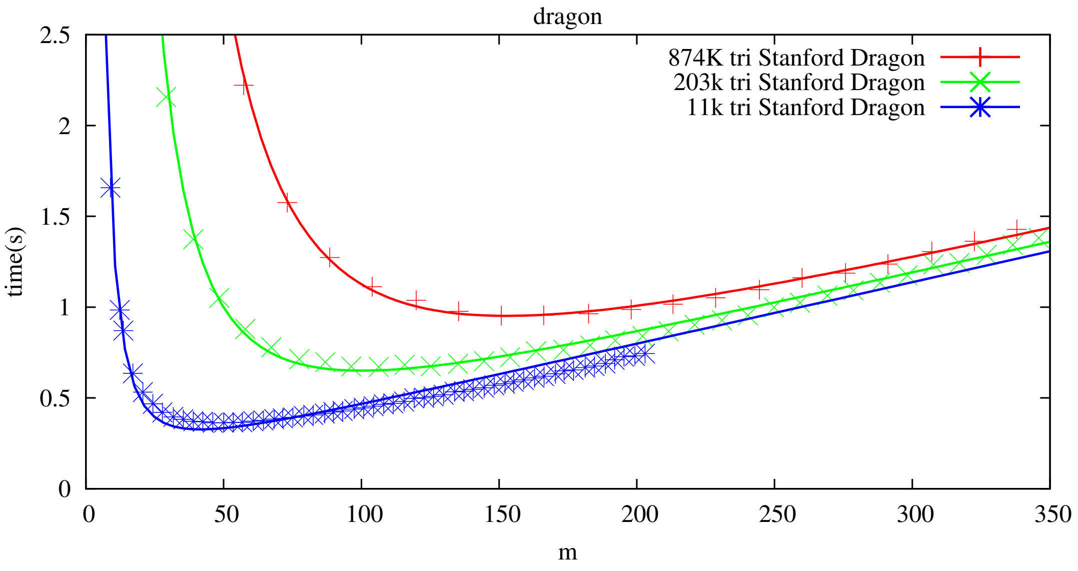


Figure 4.2. Fitted runtimes. The fitted cost runtimes (Equations 4.1 and 4.5) match very closely with the actual runtimes.

Table 4.1. M accuracy: one-level grid. The performance of $M = 8N$ for several models. Penalty is the percentage of extra time used by choosing the theoretical M versus the empirically determined optimal M.

model	N	empirical M	theoretical M	penalty
bunny	948	29172	7584	4%
	69451	645028	555608	2%
buddha	15536	132848	124288	1%
	1087716	4873932	8701728	3%
conference	282664	1121514	2261312	5%

and the resulting derivative:

$$\frac{dT}{dM} = T_{\text{step}} - \frac{N}{m^2} T_{\text{intersection}} \tag{4.6}$$

which when set to zero implies that T is minimized when:

$$m = \sqrt{N \frac{T_{\text{intersection}}}{T_{\text{step}}}} \tag{4.7}$$

and the total number of grid cells:

$$M = \left(N \frac{T_{\text{intersection}}}{T_{\text{step}}} \right)^{1.5} \tag{4.8}$$

Unfortunately, this implies memory use of $O(N^{1.5})$ but that suggests a trade-off between runtime and memory utilization that could be made by the user. Again, if we assume the ratio in the equation is 4, then we have $M = 8N^{1.5}$.

4.3.2 Empirical Test of Analysis

We have observed parts of models that have long-skinny triangles. These are typically tesselated parts that themselves are long and skinny. To mimic such situations, we have used a tesselated cylinder. The cylinder consists of very long-skinny triangles that span the entire length, with a triangle fan making up the caps. Figure 4.3 shows the 196 triangle version of the cylinder we used for testing.

Our theory suggests that $M = 8N^{1.5}$, and using that, we get good results for all of our reasonably sized tests, as shown in Table 4.2. For the extremely large 19996 triangle cylinder, using 8 is noticeably inferior and 2.3 works best. However, since it is unlikely a $O(N^{1.5})$ space-complexity algorithm would be used for a model with that many skinny triangles, for practical scenes, using 8 as the constant will work very well.

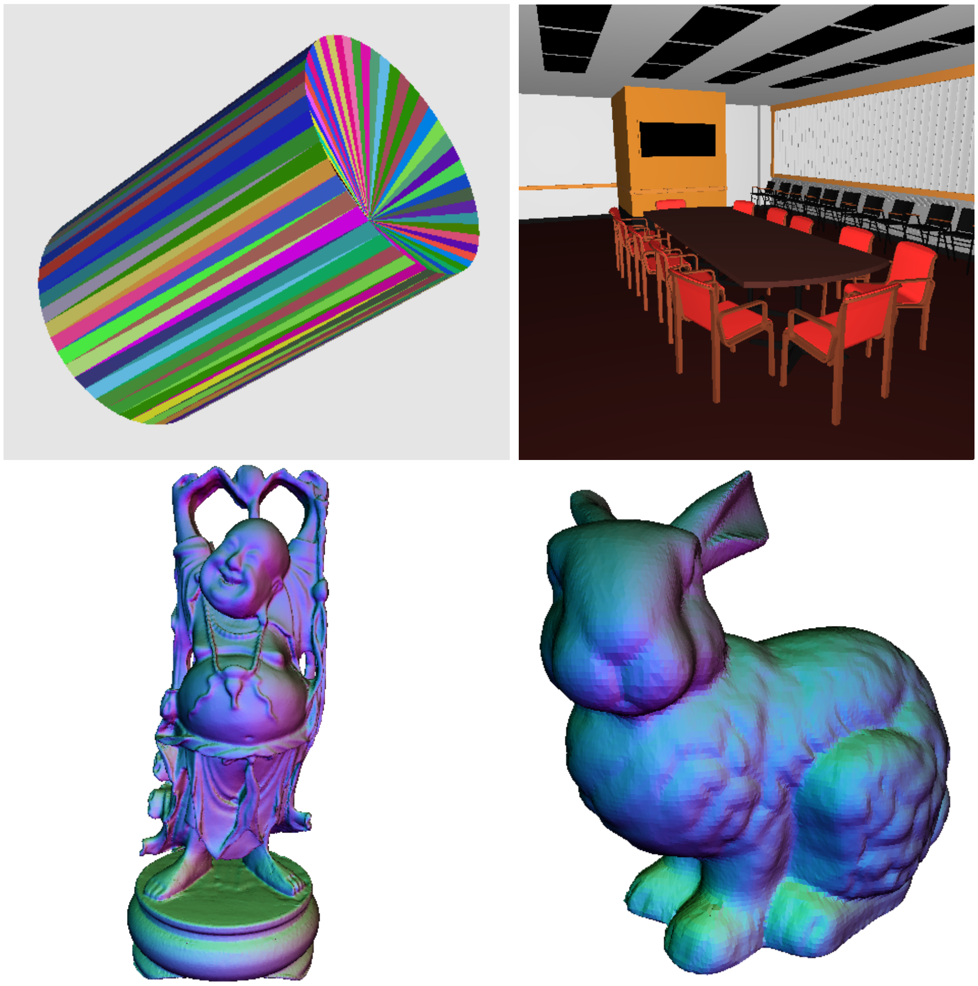


Figure 4.3. Four of the models we use in our tests. Depicted here are the 196 triangle cylinder used for modeling long-skinny triangle scenes, the 282K triangle conference room, the 1M triangle Happy Buddha statue, and the 69K triangle Stanford Bunny.

Table 4.2. M accuracy: one-level grid for long-skinny triangles. The performance of $M = 8N^{1.5}$ for a tesellated cylinder. Penalty is the percentage of extra time used by choosing the theoretical M versus the empirically determined optimal M.

model	N	empirical M	theoretical M	penalty
cylinder	196	16875	21952	0%
	3996	768337	2020822	3%
	19996	6591676	22620629	10%

Compared to using the $M = 8N$ metric that produces near-optimal single-level grids for most models, using $M = 8N^{1.5}$ as suggested by our theory produces near-optimal grids that are 1.25× faster and 14× larger for the 196 triangle cylinder, 1.7× faster and 63× larger for the 3996 triangle cylinder, and 2× faster and 141× larger for the 19996 triangle cylinder.

4.4 Multilevel Grids and Points on Surfaces

For scattered data, having hierarchical grids does little good because most or all cells will be refined, whereas for points on surfaces, there is more potential (Figure 4.4). There are a variety of ways to support a multilevel grid. Two simple options in 2D are shown in Figure 4.5. These two options trade-off geometric compactness for sharing setup costs between levels. For our analysis, we adopt the method where the whole new cell is a grid as the analysis is easy due to predictable surface area of subgrids. In implementation we adopt the geometrically compact method as in our experience it is usually faster, and it is simpler from a software engineering standpoint as black box software can be used: a grid cell can simply point to an abstract object that happens to be another grid. In addition to the two options above there are other methods that use a single high resolution grid and use some method of skipping the empty spaces, such as proximity clouds [17] and macroregions [22]. We do not try to analyze these alternatives.

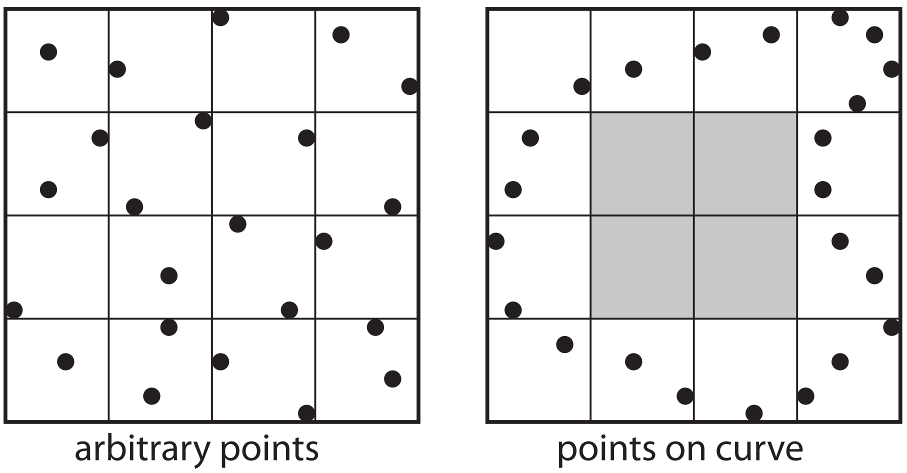


Figure 4.4. Multilevel grid for points. In scattered data most or all cells need to be refined, defeating the purpose. In a surface (curve in 2D) there are cells (shaded) that need not be refined.

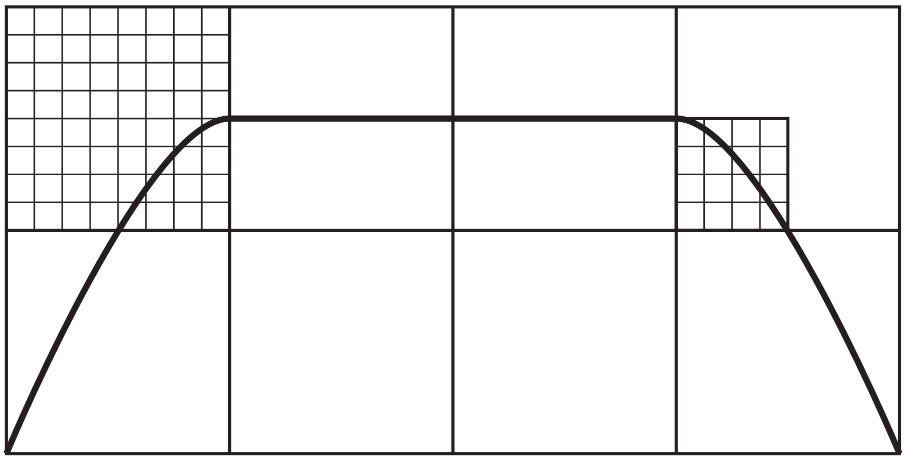


Figure 4.5. Two different ways to refine a cell.

4.4.1 Theoretical Analysis for 2-Level Grids

When a surface sweeps through a grid, it leaves many cells empty. For example, a single axis-aligned plane hits exactly one "sheet" of cells and thus occupies m_1^2 of the m_1^3 cells. A cube composed of six squares will occupy about $6m_1^2$ cells. We assume that for a given model, there will be some constant k such that the number of occupied cells is km_1^2. The rest of the m_1^3 cells are empty and would not be refined into subgrids.

A key assumption to make the analysis more tractable is that the points in occupied cells are evenly divided between those cells. Thus, each of the occupied cells has $N/(km_1^2)$ points. This assumption is usually never realistic, but we hope that as in the single-level grid analysis, a simplified case that does not discard objects will still yield useful guidance.

Since we assume the occupied cells have the same number of points, each occupied cell is divided into m_2^3 level-2 cells. Since the probability of intersecting a point is 0, a ray will pass through the entire grid, touching on the order of m_1 level-1 cells, and since the fraction of level-1 cells with a subgrid is k/m_1, the average number of subgrids hit by a ray is k. Within a subgrid, the analysis from Section 4.2 applies. Since each subgrid contains $N/(km_1^2)$ points, the average number of points in one of the cells in a level-2 grid is $N/(km_1^2m_2^3)$. For recursive grids, the setup time must be done each time a grid or subgrid is entered. Thus, the average time is:

$$T = T_{\text{setup}} + m_1 T_{\text{step}} + k\left(T_{\text{setup}} + m_2 T_{\text{step}} + \frac{N}{km_1^2 m_2^2} T_{\text{intersection}}\right) \tag{4.9}$$

If we take partial derivatives, we get

$$\frac{\partial T}{\partial m_1} = T_{\text{step}} - \frac{2N}{m_1^3 m_2^2} T_{\text{intersection}} \tag{4.10}$$

and

$$\frac{\partial T}{\partial m_2} = kT_{\text{step}} - \frac{2N}{m_1^2 m_2^3} T_{\text{intersection}} \tag{4.11}$$

Setting those both to zero and solving for m_1 yields:

$$m_1 = \sqrt[5]{N \frac{2k^2 T_{\text{intersection}}}{T_{\text{step}}}} \tag{4.12}$$

Thus, the optimal choice for the number of cells in the top level grid is:

$$M_1 = \left(N \frac{2k^2 T_{\text{intersection}}}{T_{\text{step}}} \right)^{0.6} \tag{4.13}$$

The formula for M_2 differs only in constants, and ends up being the same answer as in Section 4.2 once you account for the fact that there are $N/(km_1^2)$ points total in the subgrid. This intuitively makes sense since we want to build an optimal single-level grid for the points in the subgrid. Note that if you did the above build on scattered small points ($k = m_1$), you would get $O(N^{6/5})$ memory use and $O(N^{2/5})$ time, both of which are not ideal. The type of model matters.

4.4.2 Empirical Tests for 2-Level Grids

We first build a grid using the M_1 cells given from Equation 4.13. In each occupied cell we build an optimal single-level grid with the number of cells for that subgrid determined by Equation 4.4, where N in this case corresponds to the number of primitives in that subgrid. We use the same values of $T_{\text{intersection}}$ and T_{step} as in Section 4.2. For k we found that setting it to 1 seemed to give the best results. This is likely due to our assumptions with using points, such as a ray going all the way through the grid and not hitting any objects, not carrying over completely when using primitives. This gives us $M_1 = (8N)^{.6}$ and $M_2 = 8N_2$, where N_2 is the number of triangles in the subgrid being created.

Using just the parameters already obtained for the single-level grids (the ratio of $T_{\text{intersection}}$ to T_{step}) and setting $k = 1$, we are extremely successful in finding near-optimal 2-level grids. In table 4.3 we see that for both tesselation extremes of the buddha and bunny models, we are within 2% of the optimal time and the number of cells used appears linear. More surprisingly, even the conference room is handled by our method extremely well.

Figure 4.6 shows how varying the number of cells used for each level affects render performance. Since we use $m_1 = (8N)^{1/5}$ and $m_2 = (8N_2)^{1/3}$, this corresponds to the point $(8^{1/5}, 8^{1/3}) \approx (1.5, 2)$ in the plots, which is usually very close to the empirically

Table 4.3. M accuracy: 2-level grid. The performance of $M_1 = (8N)^{3/5}$ and $M_2 = 8N_2$ for several models. Penalty is the percentage of extra time used by choosing the theoretical M_1 versus the empirically determined optimal M_1.

model	N	empirical total cells	theoretical total cells	penalty
bunny	948	36016	17315	1%
	69451	1863727	747518	2%
buddha	15536	132848	215579	1%
	1087716	7308615	10350245	0%
conference	282664	5194622	2832557	1%

measured optimal point. For the scenes we tested, there is a large range in which to pick the parameters and still be within 5% of the measured optimal. This does not weaken the significance of our results, rather it strengthens it for two reasons. Firstly, the wide range means that large deviations in the scene, grid, and primitive intersection implementations should not cause noticeable changes in performance even if updated parameters are not determined. Or in other words, the $T_{\text{intersection}}$, T_{step}, and k terms can safely vary over a large range, showing our model to be robust to changes in these parameters. Secondly, the wide range is due to using the formulas determined by our theory; if instead a $O(N^{1/3})$ formula were used for the top level, then the valid range would be shifted for varying values of N and a range that might give good results for one scene might give poor results for another.

Compared to an optimal single ray grid, we get close to a 2× speedup in rendering time for the buddha model and a 2.4× speedup for the conference room when we use the two level grid suggested by our theory.

4.4.3 Theoretical Analysis for Multilevel Grids

Applying the logic as above, we define k_1 to be the occupancy constant for the first level grid, and k_2 for the second. Thus, for a three level grid, the cost is:

$$T = T_{\text{setup}} + m_1 T_{\text{step}} + k_1(T_{\text{setup}} + m_2 T_{\text{step}}) + k_1 k_2 \left(T_{\text{setup}} + m_3 T_{\text{step}} + \frac{N}{k_1 k_2 m_1^2 m_2^2 m_3^2} T_{\text{intersection}} \right) \quad (4.14)$$

Optimizing this shows that $m_1 = O(\sqrt[7]{N})$, and $m_2 = O(\sqrt[5]{N/m_1^2})$. More precisely,

$$M_1 = \left(2N k_1^4 k_2^2 \frac{T_{\text{intersection}}}{T_{\text{step}}} \right)^{\frac{3}{7}}. \quad (4.15)$$

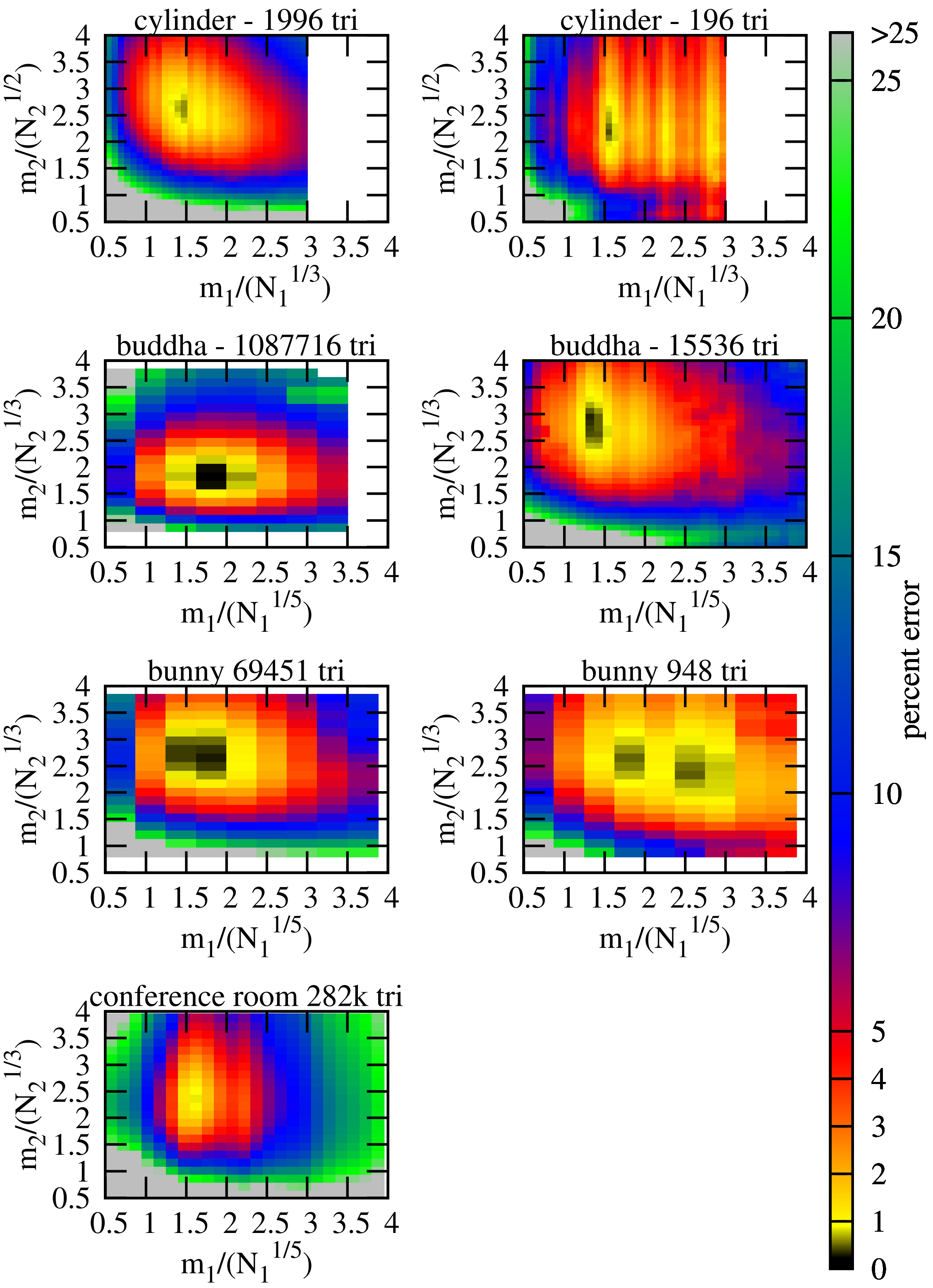


Figure 4.6. Rendering time visualization for 2-level grids. Percent error in rendering times found through an exhaustive search across the $2D$ parameter space for creating 2-level grids. Moving across the x-axis changes m_1 and the y-axis corresponds to m_2. Our theory predicts the point $(1.6, 2)$ for the cylinders and $(1.5, 2)$ for the other models as being optimal.

This suggests the general rule that for L levels in a grid, the optimal subdivision for the top level is:

$$M = O\left(N^{\frac{3}{2L+1}}\right). \tag{4.16}$$

This formula can be repeatedly applied for the number of objects in each subgrid. For example, for two levels, we would use $M_1 = O(N^{3/5})$ and then M_2 would be linear in the number of objects in that particular subgrid.

The asymptotic time complexity goes down with increasing grid levels, L, suggesting that grids with infinitely many levels would be ideal. In practice, since m is really an integral number and we are interested in finite values of N, the optimal number of grid levels is also finite. A conservative upper bound on the number of levels would likely be a multilevel grid that resembles an octree with each $m \geq 2$. A tighter bound can be obtained if we take into account the T_{setup} term. By extending Equation 4.14 to L levels and substituting in our derived optimal m terms, we obtain a cost for L levels:

$$T = T_{\text{setup}} \sum_{i=0}^{L-1} \prod_{j=1}^{i} k_j + (2L+1)\left(\frac{1}{2^{2L}} \prod_{i=1}^{L-1} k_i^{2(L-i)} T_{\text{intersection}} T_{\text{step}}^{2L} N\right)^{\frac{1}{2L+1}} \tag{4.17}$$

We can then use this equation to help determine the optimal number of grid levels to use for N compact triangles.

4.4.4 Empirical Tests for Multilevel Grids

To validate Equation 4.17, we determine the values of the T_{setup}, $T_{\text{intersection}}$, T_{step}, and k_i constant terms that give the best fit to measured render cost data. The number of triangles, N, is determined by the scene being rendered, and the number of grid levels, L, is varied from 1–6. We then empirically find the render time of a scene using an L-level grid and normalize the time in order to get a render cost that can be compared with the analytical results given by Equation 4.17. The normalizing scalar varies based on the scene being rendered; but for a given scene, it is held constant over the different values of L. The results of this are shown in Figure 4.7 and show that Equation 4.17 can be used to help predict the cost of rendering a scene with L grid levels. From this we see that a single-level grid is ideal for scenes of up to about 900 triangles, then a 2-level grid is ideal for up to about 70K triangles, followed by a 3-level grid to around 20M triangles and a 4-level grid probably until 10G triangles if there is enough physical memory. The 28M triangle Lucy model clearly shows how multilevel grids can offer a substantial benefit over single-level grids, with even the 2-level grid being over 5× faster than the single-level grid and the 4-level grid being 7× faster. The conference room is very different from our assumption of manifold-like models and yet it also closely followed our cost model, with the 3-level grid being almost 3× faster than the single-level grid.

4.4.5 Two Level Grids and Long-Skinny Triangles

As with one level grids, long, skinny objects deviate from the analysis above because they occupy a number of cells that depend on m. As before, if the primitives are on a surface, it is possible that two-level grids can improve runtime. The terms change slightly but the same basic techniques apply, yielding:

$$M_1 = kN\frac{T_{\text{intersection}}}{T_{\text{step}}} \tag{4.18}$$

and M_2, which is identical to M in Section 4.3. The associated space is:

$$\text{space} = O(N^{\frac{5}{3}}) \tag{4.19}$$

Figure 4.6 shows that our theory once again predicts to a very high accuracy what the optimal grid resolutions are. Compared to using the 2-level grid formula for regular triangles, we get a 1.6× and a 1.2× rendering speed increase for the 1996 and, respectively, 196 triangle cylinders.

4.5 Conclusion and Discussion

Clearly our analysis makes many assumptions, some of them possibly unrealistic. However, it does highlight two important things:

1. For well-behaved models, using nested grids can lower time complexity below $O(\sqrt[3]{N})$, but this comes at the cost of increased constant overhead, so the number of levels depends both on the model and on the setup cost of entering a grid. Models with more primitives, or multilevel grid implementations with low setup costs, such as with macrocells, should be able to take advantage of more levels.
2. Models with long-skinny triangles can use grids for sublinear time, but only if superlinear memory is used. If linear memory is used, then the intersection time is linear but with a reduced time constant.

The first observation above gives some explanation as to why grids can sometimes be competitive with tree-based techniques despite their worse time complexity. The difference between $N^{1/5}$ and $\log N$ requires fairly large N or large constants to become obvious.

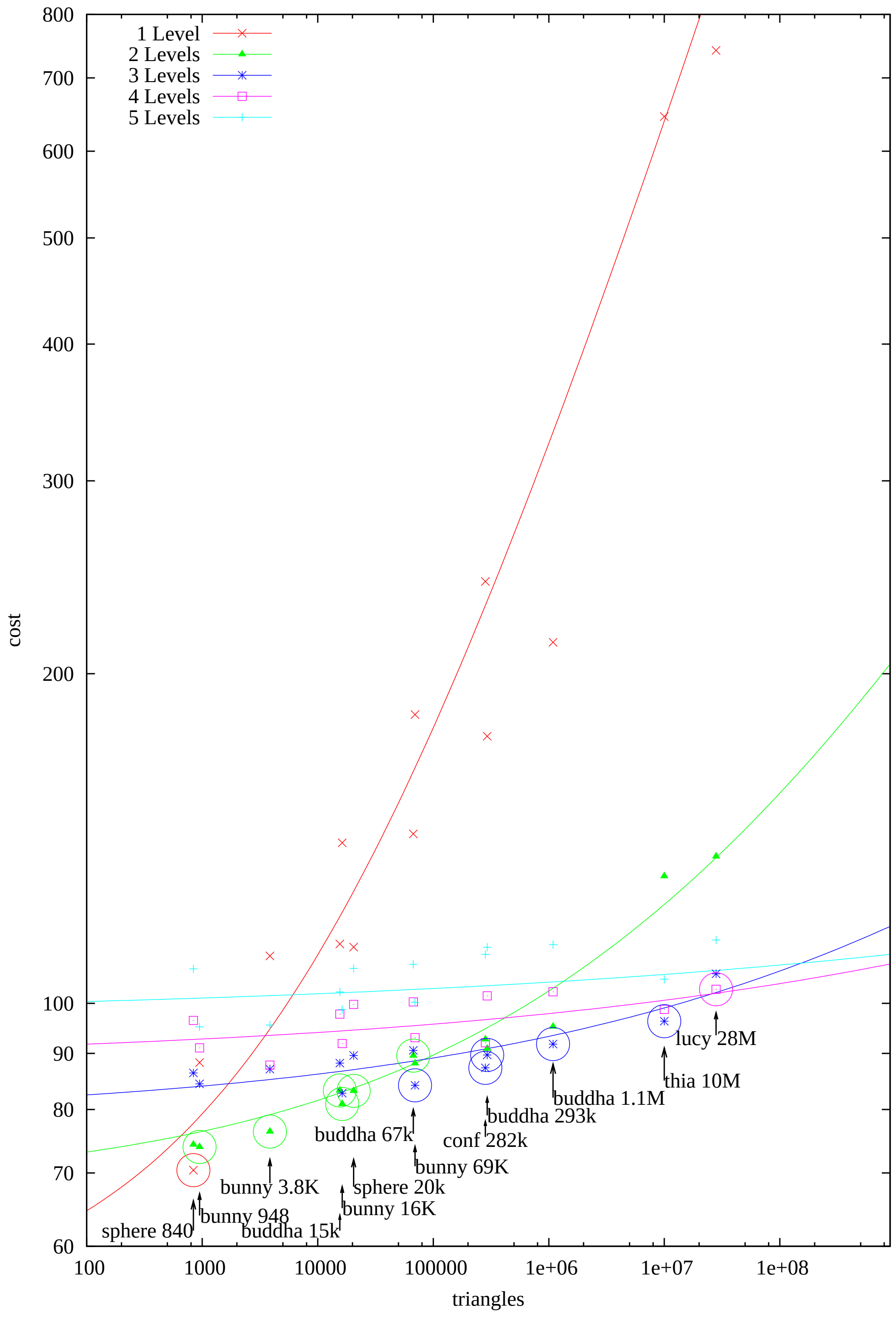


Figure 4.7. Multilevel grid cost validation. The theoretical rendering cost for multilevel grids with the experimentally determined costs of rendering various scenes. The optimal level grid for each scene is circled.

The second observation illustrates how difficult long-skinny polygons are for ray tracing in general. In fact, kd-trees and axis-aligned BVHs experience even more severe difficulties with such models [82]. It also points out that simplifying tesselated models for ray tracing is problematic—in areas of high curvature along one tangent axis and low along the other, simplified models will have long-skinny triangles that will effectively be retesselated by the grid. That all suggests oriented acceleration structures, such as the BSP in Chapter 3, should be investigated more thoroughly.

Whereas we do not recommend using the optimal M for large N, for small N, the superlinear storage cost can be better justified. Furthermore, these results can be used in picking a middle ground where performance is close to optimal, but storage use is still reasonable. This is important because many architectural or CAD-like scenes contain parts made up of long-skinny triangles. In a multilevel grid it is likely that some subgrids would consist of mainly long-skinny triangles where our results could be used.

Using mailboxing with long-skinny triangles will not allow us to use $O(N)$ grid cells since mailboxing only prevents redundant intersection tests from occurring; however, our problem is not redundant intersections, but intersecting a ray against a very large number of triangles that exist in a single cell.

Our results for determining the optimal grid dimensions for single and multilevel grids are very good for all the models we tested, which is especially surprising when we look at the simplifications we had to make. For instance, assuming the primitives are points and so have zero probability of being hit by a ray is clearly false. This could have an impact on our results since we assume a ray will always traverse through the grid and through k subgrids, when in reality, for a manifold-like object, the ray will on average hit a primitive (and thus stop traversing) after only traversing through a fraction of the grid and will usually only need to enter one or two subgrids. This would suggest that the cost to trace a ray can be, on average, significantly lower. On the other hand, since rays are hitting primitives, our assumption that the number of primitive intersections is N/m^2, which converges to 0 in the limit, is too low since at least one primitive must be hit, which raises the total cost. The intersection cost is really not even constant since missing a triangle is usually a faster code path than hitting a triangle. Finally, even just changing the camera location can affect the optimal grid resolution. In the end we expect that most of these differences are just folded into our constants, average each other out, or are insignificant. Since other unknown terms are being folded into the constants we use, this means that our constants no longer specifically represent what they started as. This means that plugging in the actual time to intersect a triangle into $T_{\text{intersection}}$ would likely lead to an incorrect solution and so these parameters should be experimentally determined.

Although our theory is corroborated very well by our experiments, it is not perfect. Interestingly enough, while experimenting with single-level grids we empirically found that using $M = O(N^{7/9})$ for manifold-like models with compact triangles and $M = O(N^{4/3})$ for manifold-like models with long-skinny triangles produced essentially perfect results for choosing M. We hypothesize that these are actually the correct formulas to use when dealing with real primitives and not points and lines. Using these results for single-level grids or the bottom level of a grid might produce even better results both in time and space costs, especially when N becomes extremely large. Proving this hypothesis would be interesting future work.

In this chapter we use an unoptimized single ray grid even though a faster grid traversal scheme exists that makes use of SIMD instructions and ray coherence [80]. We do so for two reasons. First, our analysis is based on recursive grids for which more interesting results can be proven, such as the storage complexity staying linear as the number of levels increases. The coherent grid traversal algorithm of Wald et al., on the other hand, used macro cells that do not exhibit linear storage complexity. Secondly, a single ray recursive grid is more robust than a coherent grid traverser and for scenes with incoherent rays will perform better (the coherent grid traverser would need to revert to single ray) and so the analysis still applies. Combining our results with a coherent grid traversal algorithm could produce some interesting results.

Build times have now become very important in interactive ray tracing [82]. We show that grids with compact triangles take linear space, which means building a recursive grid can be done in linear time. Although not previously mentioned, our current build times are an order of magnitude slower than those reported in Chapters 5 and 6. In addition to our build being unoptimized, it is also slower because in the other chapters we report *rebuild* time, meaning that memory has already been allocated and an initial grid built. We are confident that rebuild times of nested grids could become fast enough for handling dynamic scenes. Incorporating the rebuild time into our cost models would allow for optimizing the per frame time for dynamic scenes. One complication with doing this is that unlike with rebuild time, render time is a function of the number of rays cast and can change greatly depending on camera position, materials, lights, or ray tracing algorithm, and so without more information they cannot be directly compared with each other. One option for combining these two costs is to make some assumptions in order to predict what the render time will be. Another more interesting method would be to use the render time from the previous frame to predict how long the new frame will take to render and therefore be able to more correctly compare the build time and render time trade-offs.

CHAPTER 5

RAY TRACING FULLY DYNAMIC SCENES WITH THE COHERENT GRID TRAVERSAL

5.1 Introduction

Efficient ray-grid traversal has already received much attention [15, 28, 2, 62, 71], in aspects of both algorithm and implementation. Significant improvements cannot be expected from merely optimizing current implementations; we must explore new concepts to design an effective packetized traversal. Our algorithm delivers to grids the same components that made kd-trees and BVHs as fast as they are today: packets, SIMD extensions, and frustum traversal, while preserving the trivial computation of an incremental grid marching step. The ability to efficiently rebuild the grid on every frame enables this performance even for fully dynamic scenes that typically challenge interactive ray tracing systems.

5.2 Coherent Grid Traversal

5.2.1 Issues with Packetized Grids

The basic idea of packet and frustum traversal is straightforward: rather than traverse each ray on its own, we exploit the intrinsic coherence between neighboring rays, and trace them together. If the rays are coherent, they will largely traverse the same regions of space, accessing identical nodes in an acceleration structure, and intersecting the same underlying triangles. Effectively, the cost of memory access becomes amortized over all the rays in a packet, ideally for both our acceleration structure and geometry data. In addition, traversing multiple rays through the same node of the acceleration structure allows us to perform SIMD operations on four rays at once, reducing the computation costs of both traversal and primitive intersection by up to a factor of four. Finally, frustum techniques determine intersection patterns of an entire packet, often replacing intensive per-ray branching with a single test, thus amortizing the computations over the entire packet.

The advantages of packets, SIMD, and frustum methods are beneficial to any acceleration structure. Spatially hierarchical structures, such as a kd-tree or BVH, typically exhibit little divergence at the upper levels of traversal, making them ideally suited for adaptation to ray packets. Packets are easily traversed through hierarchical acceleration structures where rays generally progress through identical cells, diverging only in finer nodes deep down in the hierarchy, if at all. Even when rays diverge, some rays just traverse a few cells that they would not have traversed otherwise, but do not interfere with traversal decisions in the remaining part of the subtree. Since the packet is never divided, those rays automatically are re-enabled as soon as the recursion returns from that subtree.

For a grid, in contrast, the situation is more complicated: traversal is always performed on the same fine level, where divergence is most likely. Moreover, grid-based ray tracers typically use 3D digital differential analyzers (3DDDA) or Bresenham-like algorithms to iterate through the voxels traversed by the ray (e.g., [28, 2, 70]). These algorithms can chose only one cell at a time to step into, but different rays can disagree on the next cell to be traversed. For example, Figure 5.1 shows five rays diverging in cell B; some demand traversal to C, while others demand traversal to D. If the packet decides to go to C first, the 3DDDA state variables for those rays entering cell D become invalid (and vice versa). These invalid state variables break the 3DDDA algorithm in the next traversal step.

This disagreement could be solved by splitting the packet into subpackets with the same traversal decision. However, Figure 5.1 shows that the rays that have diverged in cell B still traverse other common cells (E and F) later on. If the packet were split at cell B, that coherence would be lost; in practice, packet splitting quickly deteriorates to single-ray traversal. Re-merging the packets after each step would solve that problem, but is prohibitively expensive.

5.2.2 A Slice-based Packet Traversal for Grids

As the above discussion has shown, the primary concern with packetizing a grid is that with a 3DDDA, different rays may demand different traversal orders. This is solved by abandoning 3DDDA altogether, and using an algorithm that traverses the grid *slice by slice* rather than cell by cell. For example, the rays in Figure 5.1 can be traversed by traversing through vertical slices; from cell A in the first slice, the rays are traversed

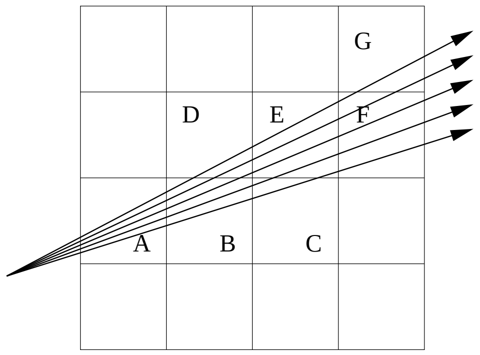


Figure 5.1. Ray divergence in a grid. Five coherent rays traversing a grid. The rays are initially together in cells A and B, but then diverge at B where they disagree on whether to first traverse C or D in the next step. Even though they have diverged, they still visit common cells (E and F) afterword.

to cells B and D in the second slice, then to C and E in the third, and so on. In each slice, all rays would intersect all of the slice's cells that are overlapped by any ray. This may traverse some rays through cells they would not have intersected themselves, but will keep the packet together at all times. In Figure 5.1, we would intersect 7 cells with 5 rays each, instead of 27 cell visits if the rays are traced individually. Though the packet now intersects only 7 instead of 27 cells, the total number of ray-cell intersection tests is $7 \times 5 = 35$. In practice, ray coherence easily compensates for this overhead.

The rays are first transformed into the canonical grid coordinate system, in which a grid of $N_x \times N_y \times N_z$ cells maps to the 3D region of $[0..N_x) \times [0..N_y) \times [0..N_z)$. In that coordinate system, the cell coordinates of any 3D point p can be computed simply by truncating it. Then the dominant component (the $\pm X$, $\pm Y$, or $\pm Z$ axis) of the direction of the first ray is picked. This will be the *major traversal axis* that we call $\vec{K}$; all rays are then traversed along this same axis; the remaining dimensions are denoted $\vec{U}$ and $\vec{V}$. In order to traverse the rays front to back, which allows early termination when all rays have intersected before the next slice, all rays must have the same sign along the traversal direction. For coherent packets, this is not a limitation; to violate this assumption, two rays would need to span an angle of more than $\frac{\pi}{2}$. Note that we do *not* demand that all rays in a packet have the same dominating axis, nor that their direction signs match along $\vec{U}$ or $\vec{V}$, as is usually required by kd-tree packet traversers [75] as long as the rays are coherent.

Now, consider a slice k along the major traversal axis, $\vec{K}$. For each ray r_i in the packet, there is a point p_i^{in} where it enters this slice, and a point p_i^{out} where it exits. The axis aligned box $\mathcal{B}$ that encloses these points will also enclose all the 3D points—and thus, the cells—visited by at least one of of the rays. Once $\mathcal{B}$ is known, truncating its min/max coordinates yields the u, v extents of all the cells on slice k that are overlapped by any of the rays (Figure 5.2).

5.2.2.1 Extension to frustum traversal. Instead of determining the overlap $\mathcal{B}$ based on the entry and exit points of *all* rays, we can compute the four planes bounding

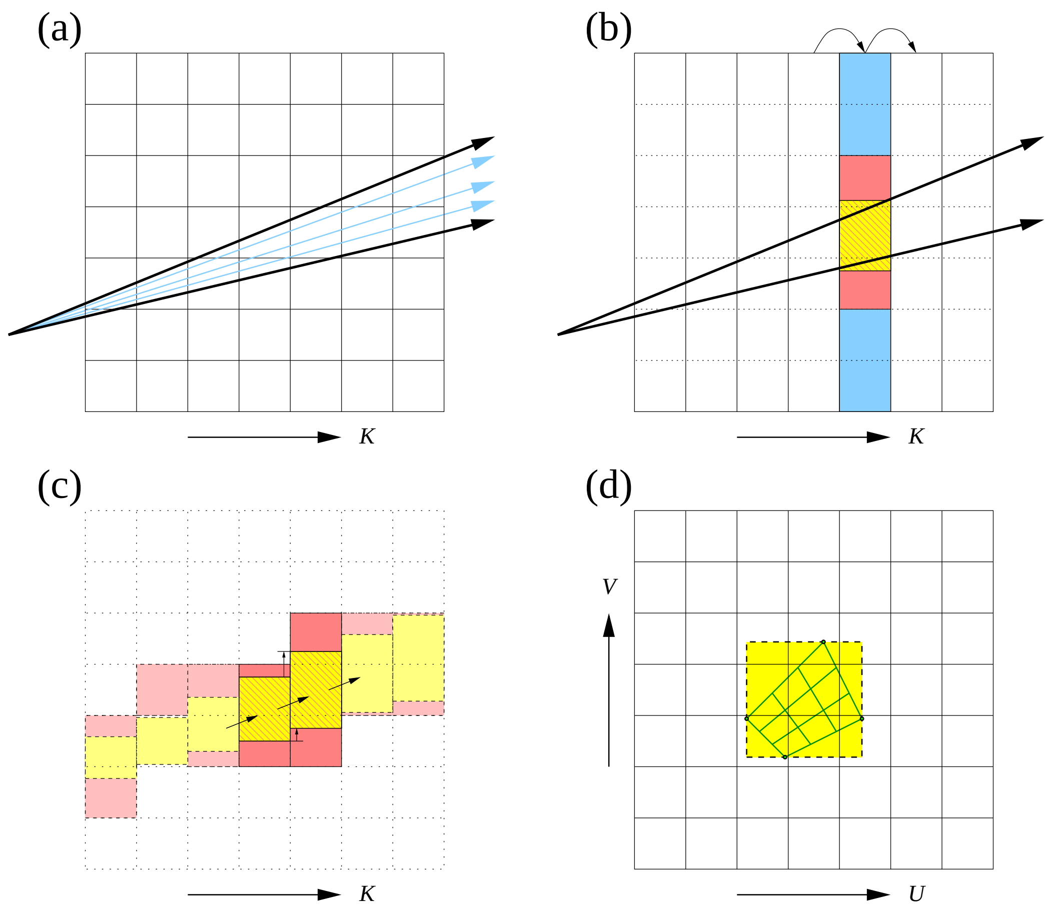


Figure 5.2. Coherent grid traversal. Given a set of coherent rays, the coherent grid traversal first computes the packet's bounding frustum (a) that is then traversed through the grid one slice at a time (b). For each slice (blue), the frustum's overlap with the slice (yellow) is incrementally computed, which determines the actual cells (red) overlapped by the frustum. (c) Independent of packet size, each frustum traversal step requires only one four-float SIMD addition to incrementally compute the min and max coordinates of the frustum slice overlap, plus one SIMD float-to-int truncation to compute the overlapped grid cells. (d) Viewed down the major traversal axis, each frustum will have a 2D overlap box whose coordinates (conservatively) define the cells overlapped by the ray packet.

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

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

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

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

Given an overlap box $B^{(i)}$, the next slice's overlap box $B^{(i+1)}$ is incrementally computed via $B^{(i+1)} = B^{(i)} + \Delta B$. This requires only four floating point additions, and can be performed with a single SIMD addition. Once a slice's overlap box B is known, the range $[i_0..i_1] \times [j_0..j_1]$ of overlapped cells can be determined by truncating $\mathcal{B}$'s coordinates and converting them to integer values. This operation can also be performed with a single SIMD float-to-int conversion instruction. Thus, for arbitrarily sized packets, the whole process of computing the next slice's overlapped cell coordinates costs only two instructions: one SIMD addition, and one SIMD float-to-int conversion. The complete algorithm is sketched in Figure 5.2.

5.2.3 Efficient Slice and Triangle Intersection

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

5.2.3.1 SIMD frustum culling. A grid does not conform as tightly to the geometry as a kd-tree, and thus requires some triangle intersections that a kd-tree would avoid (see Figure 5.3). To allow for interactive grid builds, cells are filled if they contain the bounding boxes of triangles rather than the triangles themselves, further exacerbating this problem (see Section 5.3). However, as one can see in Figure 5.3, many of these triangles will lie completely outside the frustum; had they intersected the frustum, the kd-tree would have had to perform an intersection test on them as well.

For a packet tracer, triangles outside the bounding frustum can be rejected quite cheaply using the "SIMD shaft culling" of Dmitriev et al. [24]. If the four bounding rays of the frustum miss the triangle on the *same* edge of the triangle, then all the rays must miss that triangle. Using the SIMD triangle intersection method outlined in [75], intersecting the four corner rays costs roughly as much as a single SIMD 4-ray-triangle intersection test. As such, for an N-ray packet, triangles outside the frustum can be intersected at $\frac{4}{N}$ the cost of those inside the frustum.

5.2.3.2 Mailboxing. In a grid, large triangles may overlap many cells. In addition, since a single-level grid cannot adapt to the position of a triangle, even small triangles often straddle cell boundaries. Thus, most triangles will be referenced in multiple cells. Since these references will be in neighboring cells, there is a high probability that the

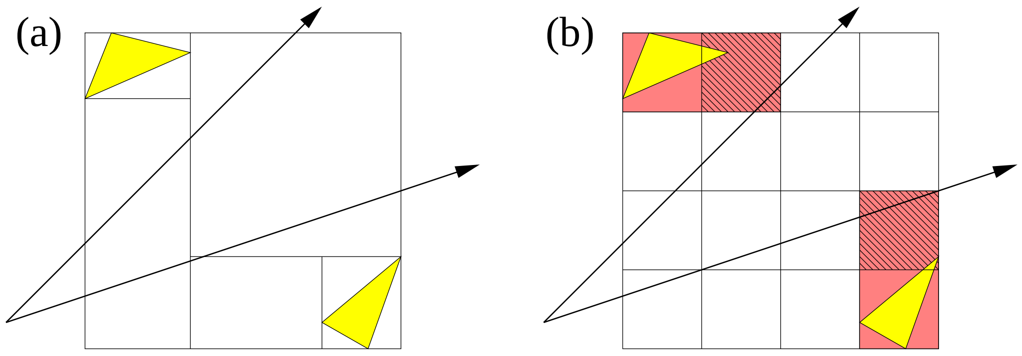


Figure 5.3. Frustum culling. Since a grid (b) does not adapt as well to the scene geometry as a kd-tree (a), a grid will often intersect triangles (red) that a kd-tree would have avoided. These triangles however usually lie far outside the view frustum, and can be inexpensively discarded by inverse frustum culling during frustum-triangle intersection.

frustum will intersect the same triangle multiple times. In fact, as shown in Figure 5.4, this is much more likely for frustum traversal than for a single-ray traversal; whereas a single ray would visit the same triangle only along one dimension, the frustum can also visit it multiple times inside the same slice.

Repeatedly intersecting the same triangle can be avoided by mailboxing [49]. Each packet is assigned a unique ID, and a triangle is tagged with that ID before the intersection test. Thus, if a packet visits a triangle already tagged with its ID, it can skip intersection. Mailboxing typically produces minimal performance improvements for single ray grids and other acceleration structures, when used with inexpensive primitives such as triangles [32]. As explained above, however, the frustum grid traversal yields far more redundant intersection tests than other acceleration structures and thus profits better from mailboxing. Additionally, the overhead of mailboxing for a packet traverser becomes insignificant; the mailbox test is performed *per packet* instead of per ray, thus amortizing the cost as we have seen before.

5.2.3.3 Impact of mailboxing and frustum culling. Mailboxing and frustum culling are both very useful in reducing the number of redundant intersection tests. In fact, both methods are much more powerful for our frustum grid traversal than for their original applications. Mailboxing is performed for multiple rays simultaneously, so the cost is amortized over the entire packet, and also avoids more redundant intersection tests.

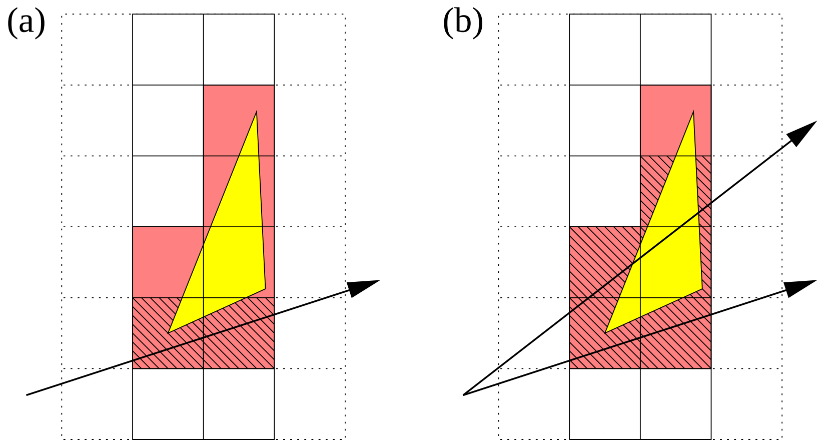


Figure 5.4. Mailboxing. While one ray (a) can revisit a triangle in multiple cells only along one dimension, a frustum (b) visits the same triangle much more often (even worse in 3D). These redundant intersection tests would be costly, but can easily be avoided by mailboxing.

Similarly, due to the higher number of redundant triangle intersections in the packetized grid, SIMD frustum culling is more beneficial than in a kd-tree, where these intersections may have been avoided in the first place.

To quantify the magnitude of this impact, we have measured statistics on example scenes, using OpenRT's kd-tree system employing 4×4 packets, and our frustum grid also using 4×4 packets. For each of those, we have measured the total number of ray-triangle intersections that are performed if neither of these techniques are used, then the results when mailboxing and finally SIMD frustum culling are applied. As can be seen from Table 5.1, mailboxing alone reduces the number of tests by up to a factor of 2; for a kd-tree, it usually trims this by less than 10% [32]. On top of the reductions achieved by mailboxing, frustum culling achieves yet another reduction by a factor of 4 to 9. With both techniques, the final number of intersection tests decreases by a factor of 8.5 to 14, and the absolute number of ray-triangle intersection tests roughly matches that of a kd-tree (see Table 5.1).

Together, mailboxing and frustum culling remedy the deficiencies of frustum traversal on uniform grids. Only one source of overhead cannot be avoided: when the bounding box of a triangle overlaps some cells traversed by a ray, but does not fall entirely outside the frustum. This scenario, however, is not limited to the grid; it also occurs in a packetized kd-tree.

5.2.4 Extension to Hierarchical Grids

Our algorithm so far has been described for a single-level grid; however, hierarchical grids generally achieve superior performance. There are several ways to organize grids hierarchically, including loosely nested grids [13, 50], recursive or multiresolution grids [43],

Table 5.1. Ray-triangle intersection tests for a 4×4 kd-tree and for our 4×4 frustum-grid traversal, and the impact of using mailboxing (MB) and frustum culling (FC). Mailboxing and frustum culling reduce the number of ray-triangle intersections by up to a factor of 14, to roughly as few as performed by a good kd-tree.

scene	#tris	grid			grid ratio	kd-tree
MB/FC		no/no	yes/no	yes/yes	no/no to yes/yes	
toys	11K	14.0M	8.7M	1.0M	14.0	0.82M
hand	15K	12.5M	6.0M	0.9M	13.9	0.85M
ben	78K	12.8M	6.0M	1.5M	8.5	1.1M
conf	274K	96.0M	54M	6.9M	13.9	3.7M

and macrocells or multigrids [61]. Though these terms are ill-defined and often used ambiguously, they all share the same idea of subdividing some regions of space more finely than others, and thus traverse empty space more quickly than populated space. To demonstrate that our approach is not restricted to uniform grids, we have extended it with a single-level macrocell layer. Macrocells are a simple hierarchical optimization to a base uniform grid, often used to apply grids to scalar volume fields [61]. Macrocells superimpose a second, coarser grid over the original fine grid, such that each macrocell corresponds to an $M \times M \times M$ block of original grid cells. Each macrocell stores a Boolean flag specifying whether any of its corresponding grid cells are occupied.

Building the macrocell grid is trivial and cheap. Frustum grid traversal with macrocells is simple; the macrocell grid in essence is just an $M \times M \times M$ downscaled version of the original grid, and many of the values computed in the frustum setup can be reused, or computed by dividing by M. During traversal, we first consider a slice of macrocells, and determine all the macrocells overlapped by the frustum (usually but one in practice). If the macrocells in the slice are all empty, M traversal steps are skipped on the original fine grid. Otherwise, these steps are performed as usual.

Though the best value of M is obviously scene dependent, and in Chapter 4 we show how multilevel grid resolutions can be more accurately chosen based on analysis, in this chapter we use a fixed value of $M = 6$, which still usually gives good results. Using macrocells yields a performance improvement of around 30%, which is consistent with improvements seen for single ray grids. Additional levels of macrocells could further improve performance for more complex models with larger grids. More robust varieties of hierarchical grids could speed up large scenes with varying geometric density, at the cost of higher build time.

5.3 Acceleration Structure Rebuild

With an animated scene, our acceleration structure is recreated every frame. Though schemes for incrementally [65] or hierarchically [53] updating a grid exist, we did not want to impose any restrictions on the kind of animations we support, and thus opted for the most general method by rebuilding the grid from scratch for every frame.

Due to having the smallest surface area in relation to volume, cubically shaped cells minimize a grid's expected ray tracing cost. Thus, we choose the grid's resolution as:

$$N_x = d_x \sqrt[3]{\frac{\lambda N}{V}}, N_y = d_y \sqrt[3]{\frac{\lambda N}{V}}, N_z = d_z \sqrt[3]{\frac{\lambda N}{V}},$$

where $\vec{d}$ is the diagonal and V the volume of our grid. We use the common scheme of choosing the number of cells to be a multiple, λ, of the number of triangles, N [15]. In Chapter 4 we show a more advanced method for determining how many cells to use for static scenes. For single level grids, both methods give the same solution and our experiments show that most scenes are insensitive to the parameter λ and achieved their best performance around $\lambda = 5$ (Figure 5.5), which we use for all the experiments throughout this paper.

Once the grid resolution is chosen, for each triangle, we determine the cells overlapped by the triangle's bounding box and add a reference to the triangle to each of these cells. Since this is quite conservative, we also tested a more exact grid insertion scheme using an exact triangle-in-box test (e.g., [1]). This conservative requirement allows the insertion step to run several times faster, while only slowing down rendering by a few percent when a mailbox algorithm with frustum traversal is employed. For instance, the conference room exhibits a wide range of triangle sizes (see Figure 3.1), and using the triangle-in-box test decreases the number of triangle references from 1.24 million to 0.83 million; however, the rendering time only improves from 339ms to 330ms, and the rebuild time almost triples from 82ms to 245ms. Thus, the total rebuild and render time become slower. One improvement is to only apply the triangle-in-box test for triangles that span many cells since they are more likely to benefit from the exact test. Yet, even if we only test triangles that span at least 200 cells, the rebuild time only improves to 118ms, which is still too slow. For scenes with dominantly long, skinny, and diagonal triangles, a more accurate test may still pay off.

Since memory allocations are costly, we use a preallocated pooled-memory scheme that prevents per-cell memory allocations and fragmentation as the scene changes from frame to frame. Memory layout techniques such as bricking [62] have also been tested; but since the frustum traversal already amortizes memory accesses over the entire packet, these techniques did not result in a measurable performance difference for our scenes. Larger grids, however, may still benefit from these techniques.

The grid clearing step is accelerated by iterating through the previous macrocells and only clearing the grid cells that have their corresponding macrocell filled. This allows us to quickly skip large empty regions without performing expensive memory reads to determine if a cell is empty. The macrocells are not explicitly cleared because they are overwritten at the end of the grid rebuild step.

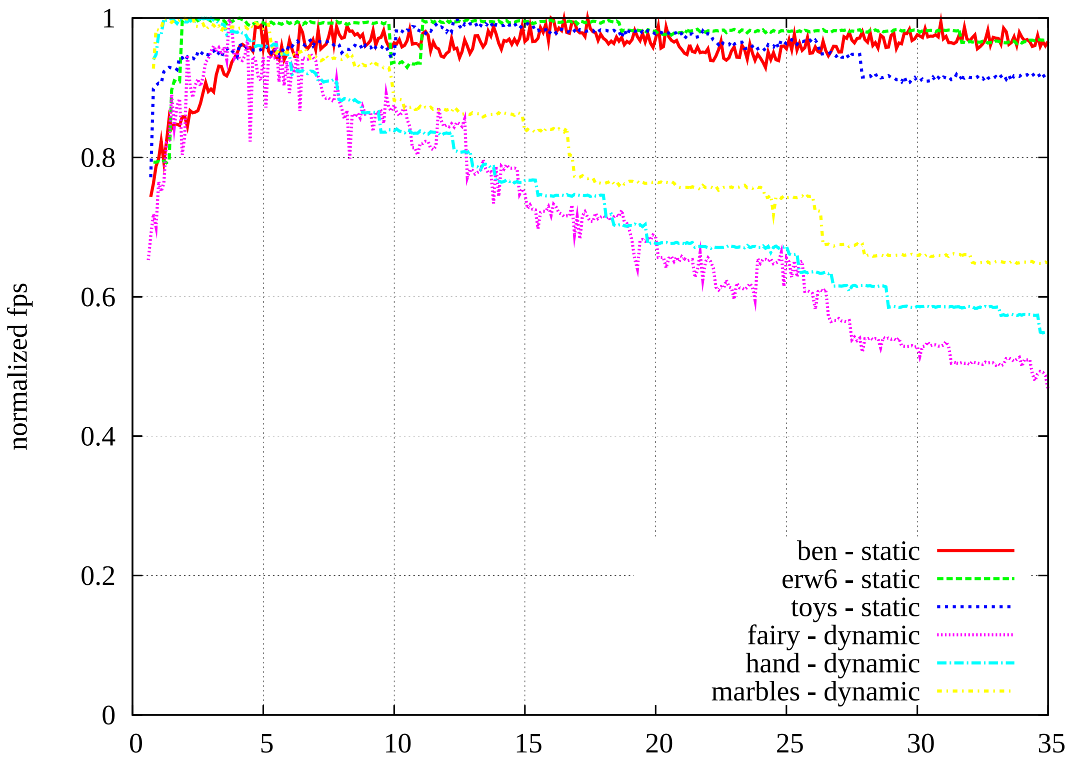


Figure 5.5. λ's affect on overall framerate. For several different models, this graph shows the framerate, normalized by the best time, in relation to grid size as determined by λ (on the X axis). Nearly all tested scenes, both static and dynamic, reach their optimum at approximately $\lambda \approx 5$.

The macrocell build time is reduced by iterating through the macrocells and checking the grid cells in each macrocell until a cell is found with a triangle. At this point, the macrocell can be marked as full before proceeding to the next macrocell.

In addition to rebuilding the grid, we also need to create the derived data for the triangle test described in [75]. Though this could be avoided by storage-free triangle tests [58, 47], we found these to be slightly inferior in performance even after per-frame triangle rebuild time is taken into account; again, this could be different for much larger scenes than we tested. Furthermore, the triangle rebuild takes less time than the grid rebuild, and can be run in parallel with the grid rebuild.

5.4 Experiments and Results

In addition to the statistics presented above, we evaluated the performance of our algorithm on a working implementation. We first discuss the impact of the different governing parameters, and present performance for both static and dynamic scenes. If not mentioned otherwise, all experiments are performed at 1024×1024 pixels, without display, and on a dual 3.2 GHz Intel Xeon PC.

5.4.1 Impact of Grid and Packet Resolution

For any given scene, the performance of our frustum traversal algorithm is governed by four factors: the resolution of the grid, macrocell resolution, screen resolution, and ray packet size. As shown in the previous section, choosing the grid resolution via $\lambda = 5$ in practice works fine for the kind of moderate sized scene we are targeting. Similar experiments show that a macrocell resolution of $6 \times 6 \times 6$ usually yields reasonable performance. Though tweaking these parameters can result in additional performance gains, these default parameters usually work well.

Although grid and macrocell resolution do have an impact, screen resolution and packet size have the greatest impact on performance. For any given packet size, the cost of a traversal step is constant, but the cost for intersecting the cells in a slice increases with the number of cells that the frustum overlaps. Larger packets will benefit more from the constant cost traversal step, but are also more likely to overlap more cells. Thus, there is a natural crossover point where the savings in traversal steps from a larger packet are offset by the additional cell intersections. Obviously, this crossover point will be influenced by the model resolution, as larger models have finer grids and correspondingly smaller cells.

To find that crossover point—and thus determine the optimal packet size—we generated different resolutions of the Stanford Armadillo model and measured the rendering performance for packets of 2×2, 4×4, 8×8, 16×16, and 32×32 rays per packet. The results of these experiments are given in Figure 5.6. For 2×2 rays, the benefit of tracing packets is rather small, and the rendering times correspondingly high. Also not surprisingly, for packets of 32×32 rays, the frusta get very wide and performance deteriorates quickly as model complexity increases. Packets of 16×16 rays are better, but still deteriorate quite quickly. For small to medium sized models, 8×8 packets performed best until the crossover point of 250k triangles, at which point the smaller 4×4 packets begin to work better for large models. If a higher degree of coherence is given for a certain application—for example for higher resolutions, multiple samples per pixel for antialiasing or motion blur, or when computing soft shadows with lots of shadow rays to the same light source—even larger packets can still be beneficial.

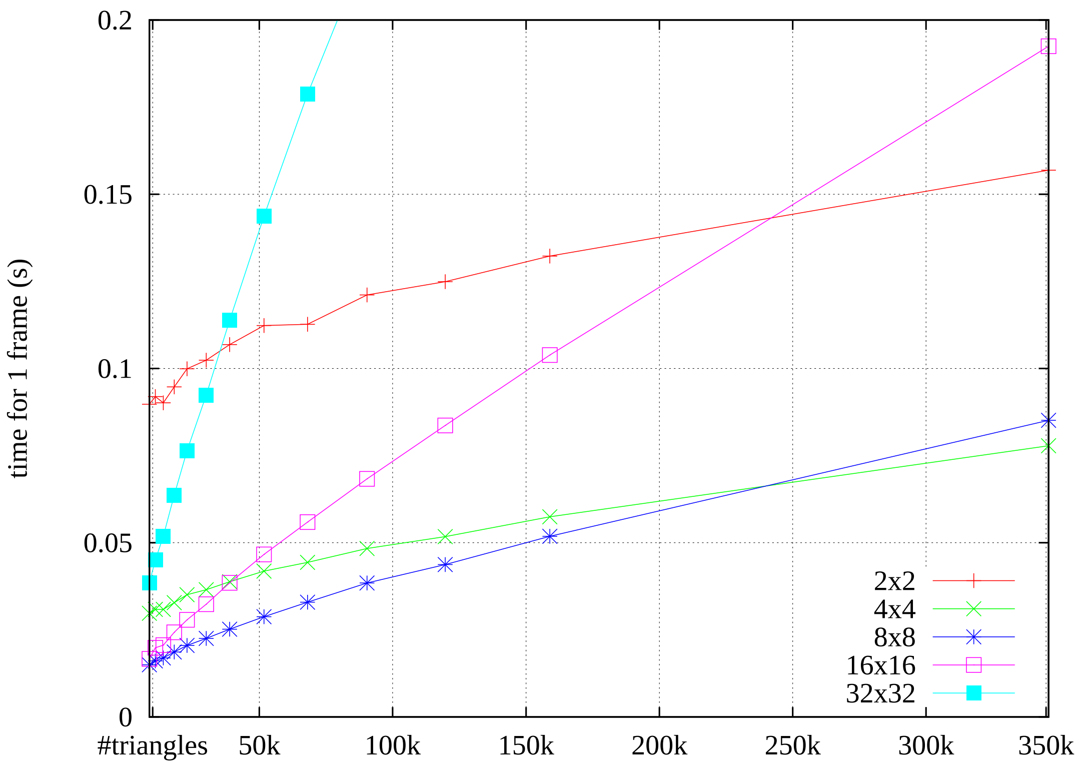


Figure 5.6. Static render time with varying packet sizes and different resolutions of the Stanford Armadillo. There is a crossover point around 250K triangles where 4×4 packets become more efficient than 8×8 packets. Nevertheless, both 4×4 and 8×8 show nearly the same performance over a wide range of model complexity.

5.4.2 Scalability with Screen Resolution

Obviously, the optimal packet size also depends on the screen resolution, as higher resolutions result in a higher density of rays, and thus allow for larger packet sizes. Given today's hardware constraints, we chose 1024×1024 pixels as a default resolution for all our experiments. In the future, high resolution displays and supersampling will push demand for even larger images.

Although the cost of ray tracing is usually considered to be linear in the number of pixels, this is not the case for our algorithm. Since higher resolutions enable larger packets, we generally see sublinear scaling in screen resolution. When increasing the screen resolution from 1024×1024 to 2048×2048 the frame rate usually drops by only a factor of 1.75–2.25, significantly less than the expected factor of 4. Weakening the linear dependence on pixel count helps overcome a major hurdle in interactive ray tracing systems.

5.4.3 Performance for Static Scenes

Though our main motivation was to enable ray tracing of dynamic scenes, the performance gains achieved by the packet traversal apply also to static models. To evaluate our raw ray tracing performance, we used several typical static test models for ray tracing (see Figure 5.7), and rendered them with our system with the rebuild disabled. This lets us consider traversal time independently from grid build time, and facilitates a comparison between our algorithm and contemporary interactive ray tracing systems, namely the frustum BVH of Wald et al. [78] and Intel's MLRT kd-tree system [67].

For this comparison, we chose the erw6, conference, and Soda Hall scenes of 800, 280K, and 2.2M triangles, respectively, as these are the only scenes for which numbers from both systems are available [67]. Though the axis-aligned features of these three architectural models strongly favor the kd-tree and BVH, Table 5.2 shows that our system, despite relatively little low-level optimization, is competitive even for these best-case scenarios for the other systems, usually being around 3–4× slower.

5.4.4 Scalability with Model Resolution

As shown in Section 5.3, for moderate-sized scenes as targeted in our system, the optimal grid resolution is usually near $\lambda \approx 5$. For significantly larger models of up to several million triangles, however, the time for building a fine grid may no longer pay off for the constant number of rays shot, and a coarser grid may yield the higher aggregate performance if build time is taken into account. As shown in Figure 5.8, for the 10 million triangle Thai Statue, the grid rebuild for $\lambda = 5$ already takes three times longer than tracing the rays. In that case, trading grid resolution for lower rebuild times pays off,

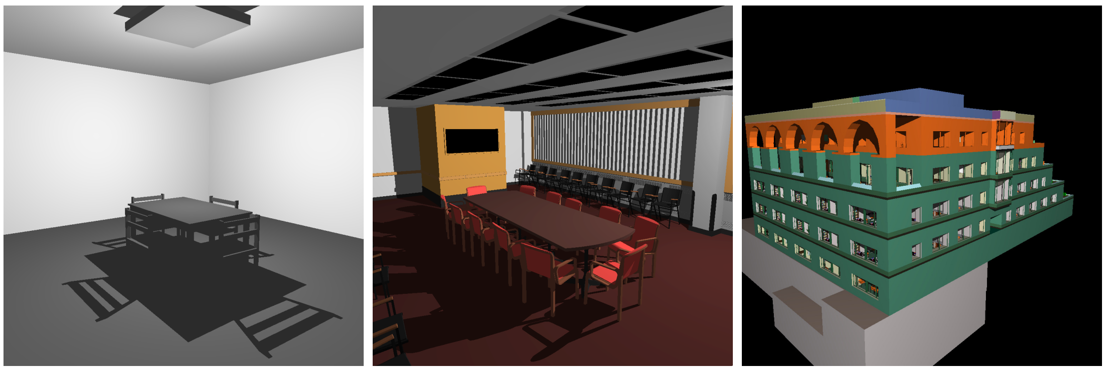


Figure 5.7. Architectural scenes. The erw6, conference, and Soda Hall scenes used for testing static scene performance.

Table 5.2. Comparison with other acceleration structures. Static scene ray tracing performance (in frames per second) for the packetized grid, BVH, and MLRT. BVH data is taken from Wald et al. and MLRT data are taken from Reshetov et al.; all times are including simple shading, but without display. Though these three scenes are best-case examples for our competitors, we remain at least competitive.

scene	#tris	BVH	MLRT	Frustum Grid
		Opteron	Pentium IV	Pentium IV
		2.6 GHz	3.2 GHz w/ HT	3.2 GHz w/ HT
erw6	804	32.5	50.7	18.3
conf	274k	9.3	15.6	4.0
soda hall	2.2m	11.1	24.1	8.0

reaching the optimal aggregate performance around $\lambda = 1$. Though the thus reduced grid resolution increases the render time, this is more than made up for in saved rebuild time, resulting in a total rendering time *including* rebuild of less then 1.5s per frame. Using a parallel build as discussed in Chapter 6 would further reduce this time.

For comparison, for the Soda Hall model, the grid at $\lambda = 1$ can be rebuilt (using one build thread only) in a mere 110ms, and achieves a frame rate (including rebuild) of 3.5 frames per second; i.e., even including rebuild, the full 2.2M triangle model is still interactive. Though this shows that a coarser grid can pay off for much larger models than intended for our technique, in the remainder of this paper we will use the above mentioned default resolution of $\lambda = 5$.

5.4.5 Comparison to Single-Ray Grid Traversal

The somewhat surprising performance of our frustum grid on architectural models can be explained by the benefits of packetization. To illustrate this difference, we compare our approach to an optimized single-ray 3DDDA implementation of a macrocell grid. Table 5.3 shows that the packetized grid ranges from 6 to 21 times faster, depending on the scene and viewpoint. Though some of this improvement is due to our use of SIMD extensions that cannot easily be used with single-ray traversal, SIMD implementation alone usually gives only about a factor of two on this hardware; the remainder is due to cost amortizations and the algorithmic improvements of the packet/frustum technique. This effect can best be explained by the number of cells visited during traversal; as we see in Table 5.4, compared to a single ray traversal, the frustum version visits roughly 10 to 20 times fewer cells for the 4×4 packets, and over 50 times fewer for the 8×8

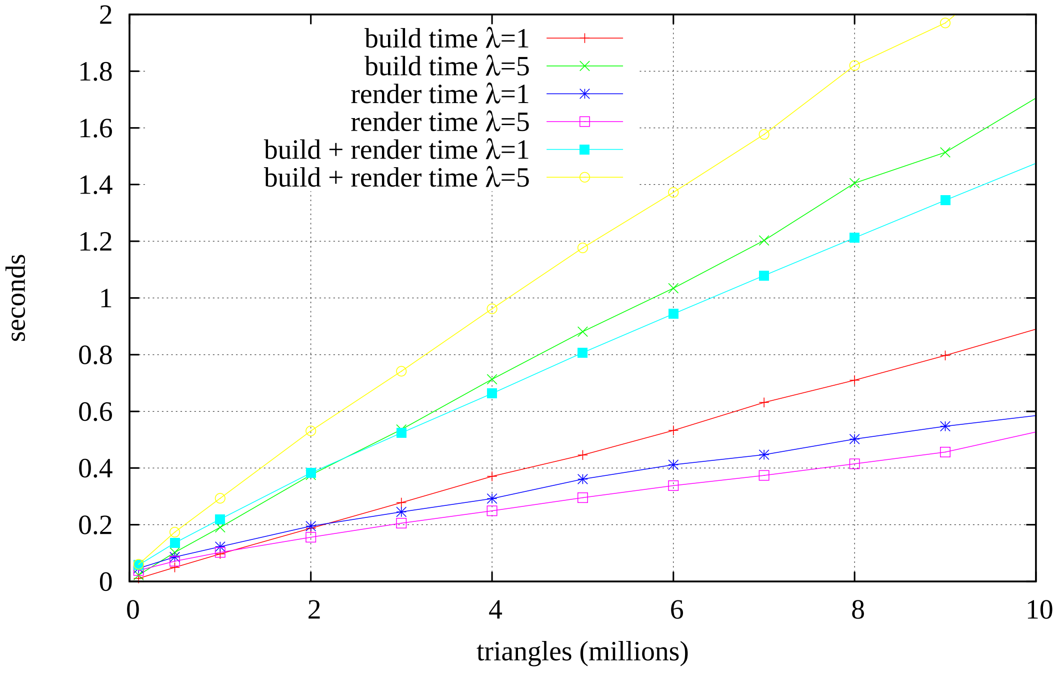


Figure 5.8. Performance with varying model resolution. Rebuild and render times at $\lambda = 1$ and $\lambda = 5$ for different resolutions of the Stanford Thai Statue ranging from 100K to 10M triangles. For these large models, we use a packet size of 4×4, although 2×2 packets perform better for the Thai Statue models larger than 3M triangles.

Table 5.3. Static scene performance (in frames per second) for our system, and for an optimized 3DDDA single-ray grid, using a macrocell hierarchy if advantageous. Images rendered at 1024×1024 pixels on a Pentium IV 3.2 GHz CPU with 1 thread and simple shading. Our frustum traversal outperforms the single-ray variant by up to an order of magnitude.

scene	ben	hand	toys	erw6	conf
single-ray	1.57	1.59	1.53	0.67	0.30
8×8 packets	10.6	16.1	20.0	14.0	3.2
ratio	6.75	10.1	13.1	20.9	10.6

packets. Due to efficient packetized slice and triangle intersection (Section 5.2.3), the frustum actually tests fewer triangle intersections as well, and can even do that in SIMD.

5.4.6 Performance for Animated Scenes

To support animation, the simplest mechanism for a grid is to rebuild the grid structure every time the geometry changes. For small to medium sized scenes, rebuilding the grid is fast, allowing the performance achieved for static scenes to be sustained during animation.

Table 5.4. Grid statistics based on ray packet size. Total number of triangles intersected and cells visited (in millions) for a single ray grid, a 4×4, and an 8×8 packet traversal. No macrocells are being used by either grid, and tests use identical dimensions for the same scene. Frustum traversal dramatically reduces both the numbers of cell visits and triangle intersection tests.

scene	ben	hand	toys	erw6	conf
# ray-triangle intersection tests (millions)					
single ray	2.96	3.58	1.97	8.90	15.70
packet 4×4	1.50	0.93	1.02	1.54	6.90
packet 8×8	5.74	2.54	2.23	2.00	20.70
# visited cells (millions)					
single ray	24.30	19.60	7.72	33.20	167.70
packet 4×4	2.91	0.95	0.80	2.18	16.54
packet 8×8	1.37	0.36	0.32	0.58	5.84
ratio 4×4	13.10	20.74	9.65	15.23	10.13
ratio 8×8	8.35	54.90	23.9	55.7	28.70

For larger scenes, other techniques such as incremental or parallel rebuilds may be required to maintain interactive performance, although these techniques were not employed in this chapter. To demonstrate these performance characteristics, we used several animated scenes of various sizes and different dynamic behavior, and measured the rebuild time and rendering performance.

5.4.6.1 Animated meshes. Some of the benchmark scenes are depicted in Figure 5.9: The "wood-doll" is a simple model with 5k triangles, resulting in a grid of $18 \times 48 \times 36$ cells that can be built in 1ms. Without shading, this scene can be rendered at 67 frames per second; even including shading and shadows, 35 frames per second can be reached. However, consisting only of rigid body animation of its otherwise static limbs, the wood-doll could also be rendered using rigid-body animation schemes for kd-trees as proposed in [77].

To stress more complex kinds of animation, we also tested an animated "hand" model of 16K triangles, as well as "ben," a runner character of 80K triangles. Though the "ben" model is already nontrivial in size, its grid of $48 \times 108 \times 78$ cells can be rebuilt in only 14ms, resulting in a final performance of 16fps without shading, and 9fps with shading and shadows turned on. The "hand" ($72 \times 36 \times 36$ cells built in 5ms) can be rendered at 36 and 16 frames per second, respectively.

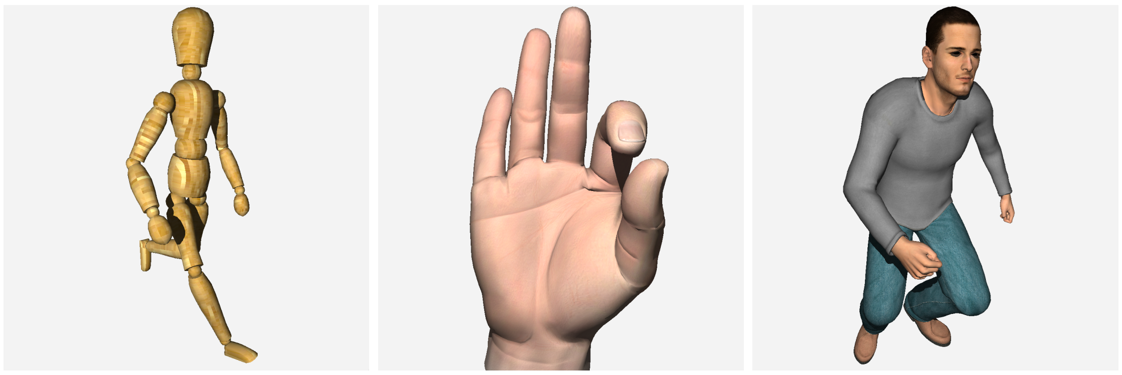


Figure 5.9. Simple animated scenes. A rigid-body wood-doll (5.3k triangles), a gesturing hand (16k triangles), and a running poser figure (78k triangles). Without shading and shadows, these scenes render at 66.9, 35.9, and 16.3 frames per second (including grid rebuild), and still at 35.1, 15.9, and 8.9 frames per second with shading, texturing, and shadows turned on.

5.4.6.2 Non-hierarchical animation. Though differing in their forms of animation, both "wood-doll," "hand," and "ben" are individual models that are tightly enclosed by the grid. To demonstrate that our method is not limited to such models, the "toys" scene has a set of 5 individually animated windup toys that walk around incoherently, bump into each other, and even jump over each other (see Figure 5.10). With a total of 11K triangles, grid rebuild (for $66 \times 18 \times 66$ cells) took 4ms, yielding a frame rate of 9–17 and 28–40fps with and without shading and shadows, respectively.

The grid's strongest advantage over other dynamic data structures is that it does not require any kind of a hierarchy to be present in the model. Thus, it can also be used for completely incoherent motion of triangles, such as explosions, physics-driven simulations, or particle sets. To demonstrate this, we modelled a scene where 110 "marbles" are dropped into an (invisible) glass box, where they participate in a rigid-body simulation (Figure 5.10). Since the grid does not depend on any kind of coherence in the motion, this kind of animation can be supported easily, taking just 2ms to rebuild ($24 \times 78 \times 24$ cells), and rendering at 20–24 and 42–50fps, respectively.

5.4.6.3 A real-world example. Although all these scenes are more or less artificial test models, the "fairy forest" scene (see Figure 5.10) has been chosen in particular because of its similarity to typical interactive scenarios. In this scene, a fairy and a dragonfly dance through an animated forest; both fairy and dragonfly are animated via a skinned skeleton. The scene incorporates both locally dense and largely empty regions; it is rather wide in spatial extent, requires complex shading, and consists of a total

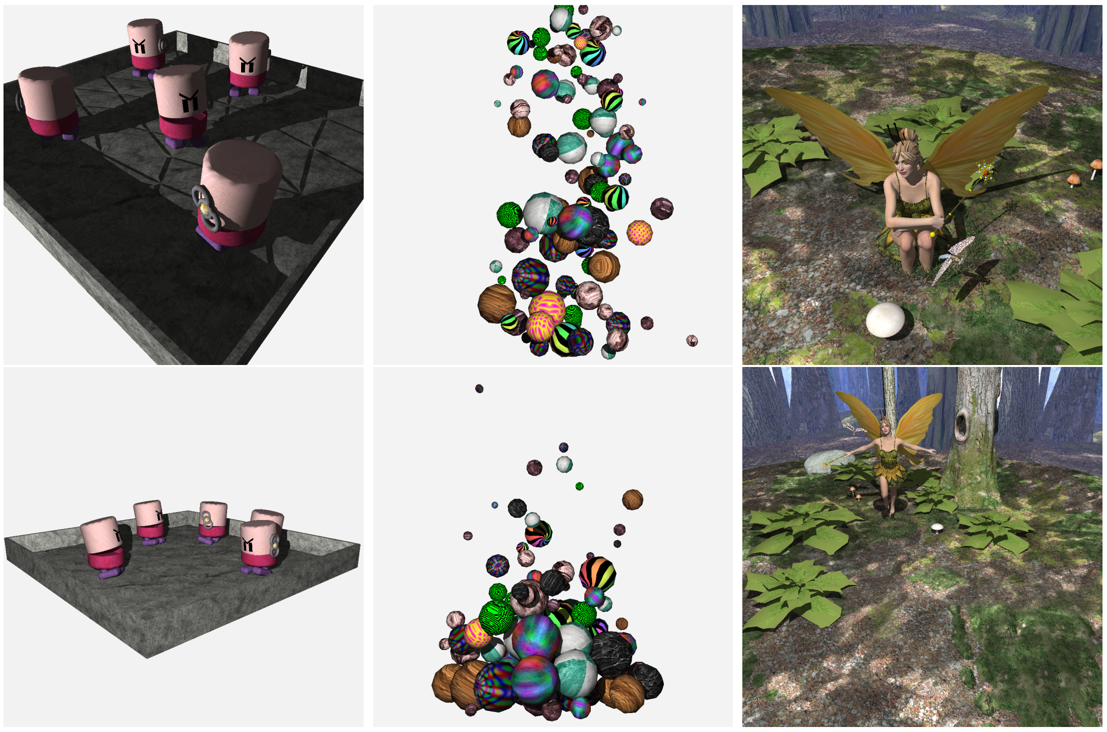


Figure 5.10. Complex animated scenes composed of multiple individual objects: a) windup toys walking around and colliding with each other (11K tri), b) a simulation of 110 marbles dropping into an (invisible) box (8.8K tri), c) a complex scene of a typical game scenario; a skinned fairy and dragonfly dance through an animated forest (174K tri total). For the camera and light positions shown, these animations respectively run at 28.0/39.6, 41.5/50.2, and 3.3/4.3 fps without shading, and still at 9.4/17.3, 19.6/24.2, and 1.3/1.8 fps if shading, texturing, and shadows are turned on.

of 174K triangles, most of which are animated. Initially, we expected the high variation in scene density to be quite a challenge for our approach. However, the frustum traversal did surprisingly well, and still achieved some 3–4 and 1–2fps for shading and no shading, respectively. The fairy's grid of $150 \times 42 \times 150$ cells can be rebuilt in 68ms.

The scenes discussed above were all modeled offline as animation sequences. This fact is not exploited at all by our traverser. The grid itself is built from a list of triangles and vertex positions every frame, neither knowing nor caring where they originate. It does not exploit the temporal coherence properties of sequenced animation, but therefore also does not depend on it. Thus, the system would work just as well for completely dynamic models. The number of triangles in the scene can easily be changed from frame to frame, and there is no restriction on the movement of existing triangles.

5.4.7 Shading, Shadows, and Secondary Rays

So far, all results have considered primary rays only. However, the true beauty of ray tracing—and its main advantage over algorithms like Z-Buffering—is that it can employ secondary rays to compute effects such as shadows, reflections, and refraction.

Among all kinds of secondary rays, shadow rays are arguably the easiest one, as they usually expose the amount of coherence that packet and frustum-based techniques like ours depend on. For most rendering algorithms, coherent shadow rays can be generated by connecting all of the primary rays' hit points to the same point light source [84]. Though certain algorithms like Monte Carlo path tracing [45]), can exhibit incoherence even in shadow rays, most practical applications of ray tracing produce coherent shadow rays, and even global illumination has already been demonstrated with such rays [9].

If we connect all surface hit points to the same point light source, the resulting shadow packets share a common origin just like primary rays, and differ from those only in that they have no concept of "corner rays." However, one can easily determine a principal march direction of the packet, and can then construct a frustum over the packet by determining the four planes that tightly bound the rays along that direction. The four edges of this frustum can then be determined quite cheaply, and can be used to perform the SIMD frustum marching and SIMD frustum culling.

Though shadow packets often *are* coherent, there is no guarantee that this is *always* the case. For example, if a primary packet hits an object's silhouette, the 3D hitpoints can be quite distant from each other, and connecting them to the same point light yields a wide frustum for which our method breaks down. In fact, for a frustum-based technique like ours the impact of some packets getting incoherent is much worse than for pure packet-based techniques, as all the triangles in the frustum would get intersected, which might comprise large parts of the scene.

Fortunately, this case can be detected and alleviated quite easily as already proposed in [84]. If the primary hit points are too far apart (measured, for example, by the minimum and maximum hit distances along the packet), the packet can be split into two more coherent subpackets. Without packet splitting, certain scene and light configurations can easily lead to severe performance degradation for shadows, whereas with splitting, shadow rays in practice work just as well as primary rays.

More general packets that do not even share the same origin would also be possible, as long as the rays are still coherent. Initial experiments have shown that this works quite well when, for example, computing soft shadows by connecting multiple surface samples with multiple light samples on the same light source. Though packet/frustum-based systems have shown that reflection and refraction rays often work surprisingly well in packet-based renderers [10, 57, 87], no experimental data are yet available for our coherent frustum traversal technique.

5.5 Summary and Discussion

We presented a new approach to ray tracing with uniform grids. This algorithm elegantly allows for transferring the recent advantages in fast ray tracing—namely, ray packets, frustum testing, and SIMD extensions—to grids, for which these techniques had previously not been available. The frustum-based grid traversal has several important advantages. First, it has a simple traversal step, where a few SIMD operations allow for determining all the cells in a grid slice that are overlapped by the frustum. This operation has a constant cost for the entire frustum that is amortized over the entire packet of rays, and allows for a traversal step that is at least as cheap as that of a packet/frustum kd-tree. Using mailboxing and SIMD frustum culling (Section 5.2.3), our method performs roughly the same number of ray-triangle intersection tests as the kd-tree. Though our implementation is not *as* highly tuned as that of Intel's MLRT system [67], it is up to 21 times faster than known single-ray grid traversal schemes, competitive with kd-trees, and inherently supports fully-dynamic animated scenes.

Our method does possess several limitations. The very nature of using a uniform grid makes the method ill-suited for highly complex scenes with a high variation in size and density of geometry, for example, the Boeing 777 data set or the classic teapot-in-a-stadium, as demonstrated in the Fairy Forest scene. Though our macrocell technique works for most cases, for highly complex scenes multiresolution grids [62], multilevel techniques [75, 53], or separation of static and dynamic objects [65], as well as mechanisms to incrementally rebuild the grid data structure may be advantageous.

Grids still suffer from common pathological cases such as large flat areas (i.e., from architectural models) where geometry overlaps numerous cells. These situations can be handled more efficiently by today's tree-based ray tracers and therefore, kd-trees and BVHs are likely to remain somewhat more efficient for many scenes. It is also not guaranteed that our technique will perform similarly well for other kinds of secondary rays like reflection and refraction, for which the coherence can be lower than for primary and shadow rays.

Our technique may be very appropriate for special-purpose hardware architectures such as GPUs that offer several times the computational power of our current hardware platform. Though kd-trees and BVHs have been realized on both architectures, they are limited by the streaming programming model in those architectures. In contrast, a grid-based iteration scheme is a better match to these architectures, and may be able to achieve a higher fraction of their peak performance. The current method may also be appropriate for a hardware-based implementation, similar to Woop et al. [87].

The primary motivation of this approach is to enable ray tracing of dynamically deforming models. Rebuilding an acceleration structure on each frame enables ray tracing these models without placing any constraints on the motion. As this update cost is—like rasterization—linear in the number of triangles, it introduces a natural limit for the size of models that can be rebuilt interactively. The rebuild cost is manageable for many applications such as visual simulation or games, where moderate scene sizes with several thousand to a few hundred thousand polygons are common.

From the performance and efficiency standpoint, compared to kd-trees and BVHs, our coherent grid traversal is arguably the most extreme one in that it is a *pure* frustum-based technique, whereas all other approaches are mixed packet/frustum-traversal techniques (i.e., [67, 78]). Compared to a packet-based technique, a pure frustum traversal can take even better benefit from coherence *if* it exists; for example, though doubling the number of rays in the packet would increase the total number of ray-triangle intersections, the *traversal* cost would not change at all. On the other hand, with rising incoherency a pure frustum-based technique will deteriorate much more quickly than the other techniques—a single incoherent ray in a packet can significantly widen a frustum, and lead to painfully degraded performance. Similarly, the frustum alone is prone to suffer worse from triangles becoming smaller. In the worst case, the frustum will intersect all the triangles in the frustum, even if those become as small as to fall in between the raster of rays in the packet. Consequently, when comparing our approach to, for instance, the BVH-based packet/frustum technique described in [78] we typically see that both techniques usually are within a factor of $\sim 2\times$ within each others performance; the BVH usually has a slight advantage—in particular for increasingly complex scenes—but can suffer worse for intentionally designed worst-case scenes, and in addition is less general in the kind of scenes it can handle.

In summary, when ray tracing coherent rays, we believe our approach to be at least competitive with other data structures and traversal algorithms known today, while at the same time being the most general of these techniques, supporting any incoherent deformation to the scene.

CHAPTER 6

AN EVALUATION OF PARALLEL GRID CONSTRUCTION FOR RAY TRACING DYNAMIC SCENES

6.1 Introduction

There has been recent progress in interactive ray tracing of fully animated scenes by making the traversal of the grid acceleration structure an order of magnitude faster and quickly rebuilding the grid from scratch each frame [80]. The faster traversal is accomplished by using the coherent grid traversal algorithm and is described in Chapter 5. In this chapter we will focus on how the grid rebuild is made interactive not just for small models, but for large models as well. Although grid rebuild time for small to moderate scenes is fast enough that only a single processor is needed for grid rebuilding, for larger scenes, the grid rebuild time becomes the dominant cost. For example, the 10 million triangle Thai Statue required almost an entire second to rebuild on one core. This chapter describes methods for parallelizing the grid rebuild so that large scenes can be rebuilt and ray traced at interactive frame rates, and by extension, small to medium sized scenes can perform even faster. With the current trend of multicore CPUs and industry projections that parallelism is the key to performance in the future, this technique becomes especially important.

Traditional ray tracers are regarded as “embarrassingly parallel,” and in fact, the first interactive ray tracers made heavy use of this property [62, 84]; however, recent work in interactive ray tracing of dynamic scenes has made updating acceleration structures, whether grid, BVH, or kd-tree, as important as the actual ray tracing [80, 52, 78, 69]. However, updating the acceleration structures in parallel, especially for many cores and at interactive frame rates, is nontrivial and until recently had not been shown for more than 2 cores. The parallel grid construction described in this chapter scales to a large number of cores before becoming performance bound by the memory architecture and was the first truly parallel update algorithm for interactive ray tracers [41]. Since then, parallel update algorithms have been shown for both BVHs [76] and kd-trees [69].

6.2 Background

6.2.1 Dynamic Scenes and CGT

The Coherent Grid Traversal [80] algorithm is able to interactively ray trace fully dynamic scenes by rebuilding the multilevel grid from scratch for each frame and quickly traversing the multilevel grid using a frustum-bounded packet. As in Chapter 5, we use a 2-level grid consisting of a top level macrocell grid built on top of the regular grid. Because the acceleration structure is rebuilt every frame, there are no restrictions on the nature of the motion in a dynamic scene, but the grid rebuild performance becomes the performance limiting factor. The grid rebuild is extremely fast for small to medium sized scenes, but the cost is linear in the number of triangles and it becomes too slow to allow for interactive ray tracing of multimillion triangle scenes.

The grid rebuild consists of three steps: clearing the previous grid cells and macrocells of previous triangle references, inserting the triangles into the grid cells that they intersect, and building the macrocells. Details can be found in Chapter 5.

6.2.2 Parallel Computers

Ray tracers have traditionally leveraged the "embarrassingly parallel" nature for static scenes to achieve interactive frame rates, thus relegating interactive ray tracing to the domain of super computers and clusters. Recently, the combination of Moore's law and algorithmic advances have allowed for personal computers with only a single core to catch up with the first interactive ray tracers that ran on super computers. Fully dynamic ray tracers have taken advantage of this by using serial algorithms to update the scene.

6.2.2.1 Multicore architectures. Although algorithmic improvements might continue at the same rate, it is unlikely that individual CPU-core performance will do so. Instead, parallelism will most likely drive future computer performance improvements, with the greatest gains stemming from the addition of multiple cores to the processor. In fact, this is already occurring, with personal computers now (circa 2009) shipping with eight cores. This means that ray tracers, especially on future personal computers, must be parallelized in order to take advantage of the faster hardware. Although this is not a problem for static scenes, which are "embarrassingly parallel," the current algorithms for handling dynamic scenes must be parallelized and able to run on many CPUs.

6.2.2.2 Distributed memory, SMP, and NUMA. There are several ways to build a multiprocessor machine. The three most common are Symmetric Multi-Processing (SMP) systems, Nonuniform Memory Access (NUMA) systems, and distributed memory systems that are usually implemented as computer clusters. SMP and NUMA are shared memory systems in which all the processors are able to access the same memory; distributed memory systems, on the other hand, are not able to share memory at the hardware level and instead must use the relatively slow network connections and software level support. On an SMP system, the computer has a single memory controller and memory bus shared by all of the processors. For memory intensive applications, this quickly becomes a bottleneck as the number of CPUs is increased. NUMA solves this problem by splitting the system into nodes made up of both processors and memory, and connecting these nodes by a fast interconnect. Each node operates like a small SMP system, with a processor capable of very fast access to the local memory on its node, regardless of how many other nodes are in the system, whereas access to memory on another node happens at slower speeds. As long as a program is written to use only local node memory, the program should scale well on a NUMA system to any number of nodes.

Although ray tracers have been written for clusters [83], and memory can be shared through a network connection [20], the parallel algorithms in this chapter are too memory-intensive to work efficiently on clusters. We thus assume a modern system that is NUMA-based with nodes that are SMP-based. In the benchmarks below, we use a machine based on 8 AMD Opteron processors, in which each processor contains multiple (2) cores and a single memory controller.

Ideally each node should only use its local memory; but since a thread can traverse any grid cell and intersect any triangle, the grid and triangle data must be shared between nodes. We avoid memory hotspots by interleaving memory allocations in a round-robin fashion by memory page, across all the nodes that might use that memory. Otherwise, we make no particular assumptions about the topology of the underlying architecture and network.

6.2.3 Parallel Construction of Data Structures

In 1998 Bartz showed how to parallelize the construction of octrees [7] for noninteractive ray tracing, and after optimizing memory allocations, on a 16 CPU SGI Challenge, Bartz obtained a parallel build efficiency between 30–80% depending on the scene used [6]. Bartz claimed his parallel build algorithm would extend to other tree-based structures; however, it is not clear whether binary trees, such as BVHs and kd-trees, would scale as well as the eight-child octree, which partitions naturally to 8 CPUs.

More recently, Benthin parallelized a fast kd-tree build for two threads and was able to halve the one second build time using two threads for a 100K Bézier patch model [8]. However, as the number of patches decreased, the scalability also went down, and below 20K patches, adding a second thread gave almost no noticeable benefit. It is likely that adding more threads would require large models in order to continue to scale. It is also unclear whether this would perform as well with simpler triangle meshes.

Most fully dynamic ray tracing algorithms make use of either acceleration structure updates [52, 78] or full acceleration structure rebuilds [52, 80]. When the results of this chapter were first published [41], only Lauterbach et al. [52] had parallelized an update and rebuild algorithm for an interactive ray tracer. However, although their algorithms do update the bounding volumes and perform the BVH tree rebuild in parallel, the authors report that this works because their data is well balanced and they show results only for a dual core computer. Since the parallel grid rebuild of Ize et al. was published, fully parallel kd-tree [69] and BVH [76] builds have been presented. However, the builds are still not as fast as the grid build and it is unclear whether they can scale as well as the grid build.

6.3 Parallel Grid Rebuild

We rebuild the grid using the same method as presented in Chapter 5, except that now the cell clearing, triangle insertion, and macrocell building phases have been parallelized. We also distribute the triangle and grid storage by memory page across all nodes being used. We discuss the parallelization of each of these phases here.

6.3.1 Parallel Cell Clearing and Macrocell Build

We parallelize the grid clearing and macrocell building by statically assigning each thread a continuous section of macrocells. By making the assignments over continuous sections, we benefit from cache coherence and a simple iteration over the cells. Furthermore, since each thread handles its own macrocells, and by extension, grid cells, there is no need for mutex locking. Unfortunately, the static assignment and continuous allocation are subject to load imbalance because some sections of the grid may have more triangles than others. However, when we tried to improve this by scheduling small sections of cells round robin to the threads or using dynamic load balancing, the extra overhead and lack of memory locality resulted in a slowdown for the overall clearing and macrocell building phases.

6.3.2 Parallel Triangle Insertion

The triangle insertion step is equivalent to triangle rasterization onto a regular structure of 3D cells. Indeed, it is the three-dimensional extension to the traditional rasterization of triangles onto 2D pixels. As such, we can take advantage of the existing general parallel rendering methods, such as those in Chromium [35], to parallelize the triangle insertion.

In the 3D grid, a triangle that overlaps several cells will be placed in each of those cells. This is akin to the 2D rasterization of a triangle into multiple pixel fragments. The only difference is that in 2D rasterization, the fragments are combined together via z-buffering and blending ops, whereas the grid accumulates them all into a per-cell list of triangles.

Parallel rendering algorithms are classified into three categories: sort-first, sort-middle, and sort-last [59]. In sort-first, each processor is given a portion of the image. The processor determines which triangles are needed to fill that part of the image by iterating through all the triangles. Sort-last assigns each processor a portion of the triangles and determines which pixels those triangles belong in. The sort-middle approach assigns each processor a portion of the image and a portion of the triangles to work with. We can borrow these same concepts and apply them to the grid build, leading to three basic approaches to parallel grid builds.

6.3.2.1 Sort-first build. In a sort-first parallel build, each thread is assigned a subset of the cells to fill, and then it finds all the triangles that overlap those cells. For example, for T threads, the grid could be subdivided into T subgrids. Each thread iterates over all N triangles, discards those that do not overlap its subgrid, and rasterizes the rest.

The advantage of sort-first is that each cell is written by exactly one thread; there is no write combining. Since shared data is only read, there are no shared writes, and so synchronization is required only at the end of the algorithm. However, the disadvantage is that each thread has to read every triangle and at the very least, has to test whether the triangle overlaps its subgrid; hence, its complexity is $O(N + \Theta)$, where Θ is the parallelization overhead. In addition, load balancing is nontrivial when the triangle distribution is uneven.

6.3.2.2 Sort-last build. This approach is exactly the opposite of sort-first, in that each thread operates on a fixed set of triangles. The thread partitions the triangle into fragments and stores the fragments in the appropriate cells. The complexity of this algorithm is $O(\frac{N}{T} + \Theta)$ so it will scale as more threads are added.

The advantage to sort-last is that each triangle is touched by exactly one thread, and this method can easily be load balanced. The drawback is the scatter write and write conflicts. Any thread can write to any cell at any time, and different threads may want to write to the same cell. This results in bad memory performance and is bad for streaming architectures (which usually support scatter writes poorly, if at all). In particular, the write combining—multiple threads might want to write to the same cell at the same time—requires synchronization at every write, which is bad for scalability. Alternatively, each thread could write its triangles into its own grid, and the thread-local grids could then either be merged into a single grid, without the use of synchronization, or the thread-local grids could all be individually ray traced.

6.3.2.3 Sort-middle build. Sort-middle falls in between the previously discussed approaches. Each thread is responsible for a specific set of triangles and grid cells. Each thread iterates over its triangles, partitions each one into fragments, and routes these to the respective threads that fill them into the final cells. Like the sort-last approach, sort-middle also has a complexity of $O(\frac{N}{T} + \Theta)$.

The benefits of sort-middle are that it is relatively easy to load balance, each triangle is read only once, and there are no write conflicts. There is no scatter read nor scatter write, which is good for broadband/streaming architectures. The disadvantage is that it requires buffering of fragments between the two stages. The best way to realize the buffering is not obvious.

From these three methods, we will consider the latter two in more detail. As discussed above, for nonuniform primitive distributions, sort-first is problematic to load balance. Since nonuniform distribution is certainly the case for our applications, load balancing must be addressed. In addition, our applications have a lot of triangles, and reading each triangle several times is likely to quickly produce a memory bottleneck. Thus, we will ignore sort-first and only discuss the other two options below.

6.3.3 Sort-Last Approaches

As noted above, each thread works on different triangles. We can easily do this by statically assigning each thread a section of triangles. For each triangle assigned to a thread, it must find the cells the triangle overlaps and add the triangle to those per-cell lists. As a result, all threads have the ability to write to all grid cells, which requires synchronizing writes to each cell. There are several ways to do this as described below.

6.3.3.1 Single mutex. The most straightforward approach to synchronization is to use a single mutex for inserting triangles into the cells. A single mutex results in one triangle insertion at a time, regardless of what cell it falls inside. This results in an overhead complexity of $\Theta = \frac{N}{M}$, where M is the number of mutexes. Since we are using only one mutex, we get a total complexity of $O(\frac{N}{T} + \frac{N}{1}) = O(N)$. Therefore, using the same mutex for every insertion will quickly cause the parallel build to break down into a slow serial build. Clearly this is undesirable for a parallel build, as many of the available resources are not being utilized.

6.3.3.2 One mutex per cell. Rather than share a single mutex, we can use a unique mutex for every grid cell to prevent false conflicts from occurring. The advantage of this method over the naive build is that locking a mutex for one cell will not affect any other cell in the grid. Our complexity now becomes $O(\frac{N}{T} + \frac{N}{M}) = O(\frac{N}{T} + \frac{N}{N \cdot \lambda})$, where λ is the grid resolution factor as defined in [80]. Since λ is usually constant, this simplifies to $O(\frac{N}{T})$. This method eliminates most of the lock contention, but pays a considerable cost for the size of the mutex pool and the memory accesses to it.

6.3.3.3 Mutex pool. Instead of using a single mutex per cell, it is possible to use a smaller pool of locks, each of which are mapped to multiple grid cells. Since each thread always locks only one mutex at a time, this is deadlock free. We can attempt to maximize some of the memory coherency issues for the mutexes by sharing a mutex among neighboring cells. We accomplish this by having a mutex per macrocell, which is shared by all the cells in a macrocell. In addition, if we can perform a mutex lock once per triangle instead of once for each cell the triangle overlaps, then the reduction in mutex locks would further improve performance. Given a list of cells that a triangle must be inserted into, we perform a mutex lock on the first cell, insert the triangle into the cell, and then rather than immediately releasing the lock and reacquiring it for the next cell, we check to see whether the next cell shares the same mutex as the current cell and if it does, we do not release the mutex. Once all the cells for that triangle have been visited, we release the lock. Holding a lock over several iterations of cells is not expensive since the only work done in those iterations, aside from the bookkeeping to determine which macrocell we are in, is the critical section that requires the mutex. The complexity of this approach is the same as the one mutex per cell approach.

6.3.3.4 Grid merge. Rather than have each thread write the triangles to a common grid, which requires at least one synchronization per triangle, we could have each thread write its triangles into its own thread local grid. Once the thread local grids have been filled, each thread can be responsible for merging certain sections of the local grids into the main grid. Or, rather than merging during grid build, the T grids could simply all be checked during ray traversal, as in [55, 73]; this would clearly result in a very fast triangle insertion that should scale very well, since aside from the triangle reading, all other memory operations would be local to the node, but at the expense of making ray traversal slower. Furthermore, delaying the merge would require the macrocell build to look at T grids, so the macrocell build would end up with a linear complexity. Both versions require each thread to clear the grid it built, which also results in a linear complexity for the clear stage. Like the two previous approaches, the complexity of the triangle insertion is also $O(\frac{N}{T})$, but the other stages become slower, which results in the overall build and ray traversal time becoming much slower than the other methods.

6.3.4 Sort-Middle Grid Build

In our sort-middle approach, we perform a coarse parallel bucket sort of the triangles by their cell location. Then each thread takes a set of buckets and writes the triangles in those buckets to the grid cells. Since each thread handles writing into different parts of the grid, as specified by the buckets, there is no chance of multiple threads writing to the same grid cells; thus, mutexes are not required.

More specifically, given N triangles and T threads, each thread takes $\frac{N}{T}$ triangles and sorts these triangles into T thread local buckets (see Figure 6.1 for an example with $T = 4$). We choose to sort based on the triangle's z cell index modulo the T buckets since that distributes the triangles fairly equally among the buckets, while still allowing us to exploit grid memory coherence when inserting the triangles into the grid because the cells in an xy slice are located in a contiguous section of memory. Note that a triangle may overlap multiple cells, resulting in a triangle appearing in multiple buckets. Once all the threads have finished sorting their $\frac{N}{T}$ triangles into their T buckets, we need to merge the triangles from the buckets into the grid. As there are T sets of T buckets, we do this by reordering the buckets among the threads so that thread i now becomes responsible for the ith bucket from each thread, that is the bucket whose z index is $z \bmod T = i$. Thread i then adds all these triangles into the grid. Since thread i now has *all* buckets with the

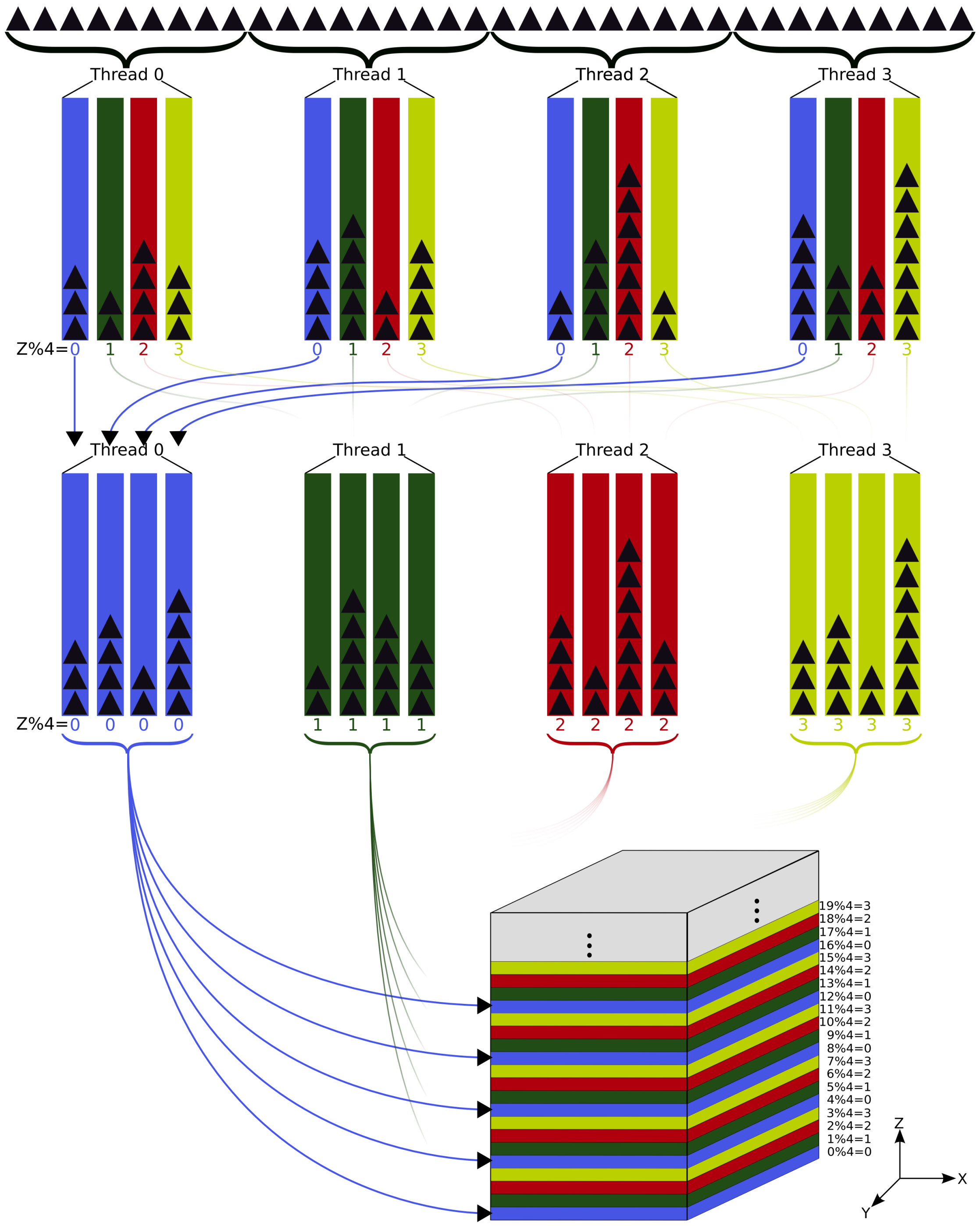


Figure 6.1. Sort-middle build. Given 4 processors, in the sort-middle build, the N triangles are equally divided among the 4 threads. Each thread has 4 buckets that are used to sort its $\frac{N}{4}$ triangles, based on the z cell index modulo 4. After all 4 threads have finished sorting, the threads regroup the buckets and fill in the grid. Thread i is responsible for the ith buckets of each of the threads and places all of those triangles into the grid. For example, Thread 0 takes bucket 0 from all 4 threads, Thread 1 takes bucket 1 from all 4 threads, and so on. Since all i buckets correspond to the same z cell indices, there is no overlap in what grid cells they are writing to, thus eliminating the need for mutexes.

same z indices (those for which $z \bmod T = i$), thread i will be the only thread writing to those grid cells.

Mutexes are not required at all as each thread is guaranteed to be writing to different grid cells. This method removes the mutex locking overhead; however, the trade off is that it adds extra work in writing to the buckets. The complexity is also $O(\frac{N}{T})$.

6.4 Results

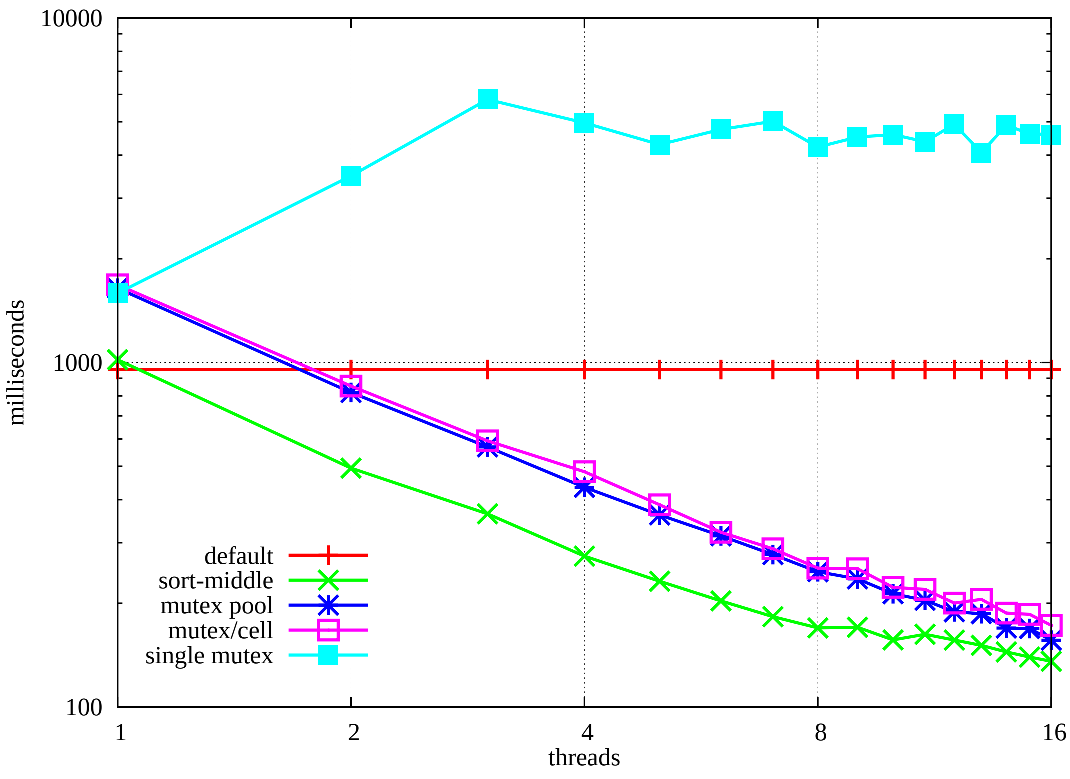
Unless otherwise noted, we use the 10 million triangle Thai statue at a grid resolution of $192 \times 324 \times 168$ and a macrocell resolution factor of 6 for our measurements.

We use a 16 core computer composed of 8 nodes, each of which has a Dual Core Opteron 880 Processor running at 2.4GHz and 8GB of local memory (a total of 64 GB). Since the cores on a processor must compete with each other for use of the shared memory controller, we distribute the threads across the processors so that a processor will only have both cores filled once all the other processors have at least one core being used. We also interleave memory across the nodes that will need to access it.

6.4.1 Comparison of Build Methods

In order to determine which build method performs best, we compare the sort-middle, sort-last, and the serial build algorithms. As expected, Figure 6.2 shows that for one thread, the serial algorithm performs best, since it has no parallel code overhead. The mutex-based builds show that the penalty for locking a mutex for every cell a triangle occupies, or even for every macrocell in the case of the mutex pool, is quite large. The single mutex build performs slightly better than the other mutex builds when there is only one thread, since the mutex is always in cache. Surprisingly, the sort-middle build performs almost as well as the serial build for one thread, despite the extra overhead of passing the triangles through an intermediate buffer.

As we increase the number of threads, we find that using a single mutex scales poorly as expected, and also degrades in performance significantly as more threads are added. This is mainly due to our mutex implementation, which requires an expensive system call when multiple threads try to lock the mutex at the same time. The mutex per cell, mutex pool, and sort-middle builds, on the other hand, do scale to the number of threads; however, since the mutex builds are initially twice as slow with one thread, they are unable to perform better than the sort-middle build. Since the mutex pool build requires fewer mutex locks, it performs slightly better than the mutex per cell build.



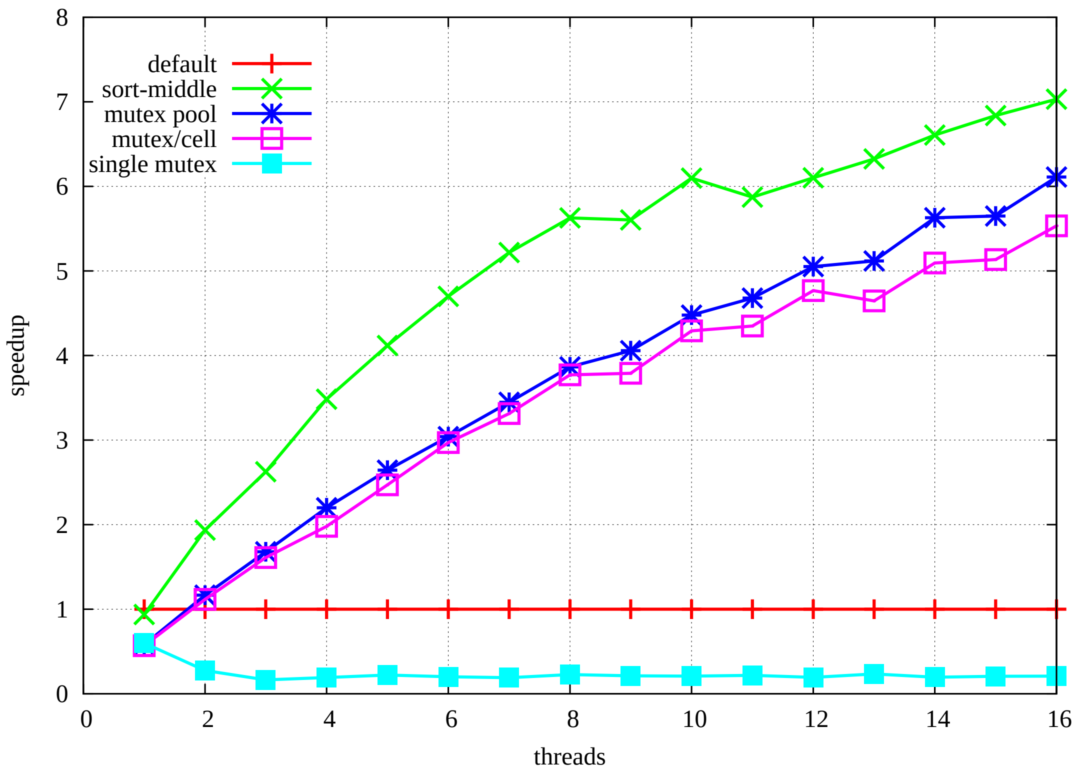


Figure 6.2. Total rebuild performance (clearing + triangle insertion + macrocell build) for the Thai Statue for: a single mutex, a mutex per cell, mutex pools, sort-middle, and the default serial rebuild.

6.4.2 Individual Steps

Examining the individual steps allows us to better understand why scalability drops as the number of threads increases. Figure 6.3 shows how long the fastest and slowest threads take to clear the grid cells, to build the macrocells, and to perform the two fastest triangle insertion builds: the sort-middle build and the mutex pool build. Increasing the number of threads would ideally cause all of the steps to scale linearly and all of the threads in a step to be equally fast. However, this is not the case due to poor load balancing and memory limitations.

We see that for the Thai statue, the grid clearing phase exhibits rather poor load balancing, with some threads finishing an order of magnitude faster than the slowest thread. This is easily explained by observing that different slices along the z axis contain varying amounts of empty cells, with the end slices being mostly empty and the middle cells being mostly full. Since an empty macrocell allows a thread to skip looking at the individual grid cells, this results in a very fast clear for some threads. We tried other load

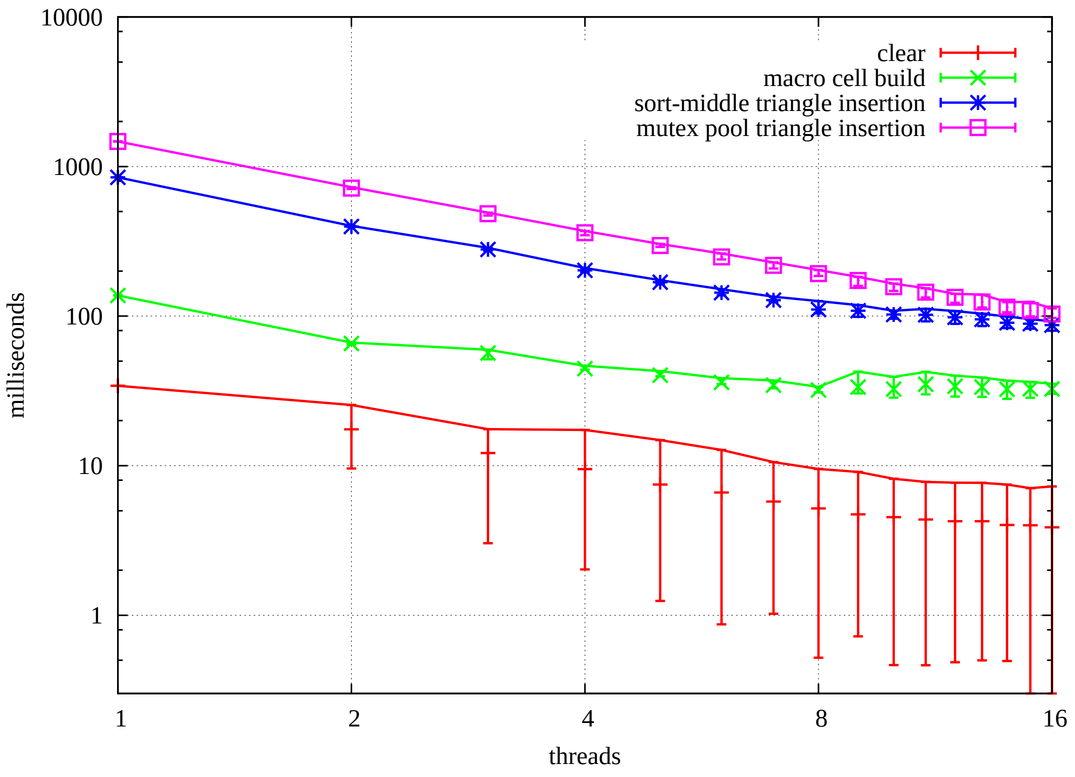


Figure 6.3. Individual build steps for Thai Statue. The min, average, and max thread times help show how the load balancing performs.

balancing methods, such as having the threads traverse the cells in strides rather than in continuous chunks, and dynamically assigning cells to threads. Although these methods did improve load balancing, the extra overhead caused the overall time to increase.

Whereas the macrocell build shares the same load balancing and a similar iteration scheme as the cell clearing, it is able to load balance fairly well up to 8 threads, after which a limit is encountered, making some of the threads much slower and the fastest threads barely faster. If this was due to poor load balancing, then we would expect a slowdown in the slowest threads to be matched by a speedup in the fastest threads, but this does not occur. Also note that the gap is largest at 9 threads, and starts to go down until 16 threads, at which point the threads all take roughly the same amount of time. This can be explained by noticing that at 8 threads, each node has only one thread on it, so each thread has an equal amount of access to all the resources; but at 9 threads, two threads will exist on one node, so those two threads must share node resources. By 16 threads, all the nodes once again have an equal number of threads (2).

The triangle insertion steps load balance fairly well, with some variation most likely due to threads having triangles that need to be inserted into more cells than usual. However, most of the variation once again comes from unequal contention of the memory resources, which can be seen from the increased range of rebuild times from 9 threads to 15 threads, with 16 threads once again having little variation.

6.4.3 Memory System Overhead

As Figure 6.3 shows, the different build stages scale at different rates and none of them scale perfectly. This is due to a combination of parallel algorithm overhead, poor load balancing, and memory system bottleneck. We can measure the memory system impact by placing mutexes around each build stage and only allowing one thread to run at a time. Figure 6.4 shows the result of this “contention-free” build. Notice that the macrocell build scales the worst when all the threads are contending for resources, but scales extremely well in the “contention-free” build. The sort-middle triangle insertion also scales extremely well, almost achieving perfect scaling, when resource contention is removed. This suggests that when all the threads are working concurrently, the memory traffic is the main bottleneck, except for the mutex builds that, in addition to being affected by the same memory penalty, also have a large parallel algorithm overhead due to the mutex locking and unlocking, which cuts the triangle insertion performance in half.

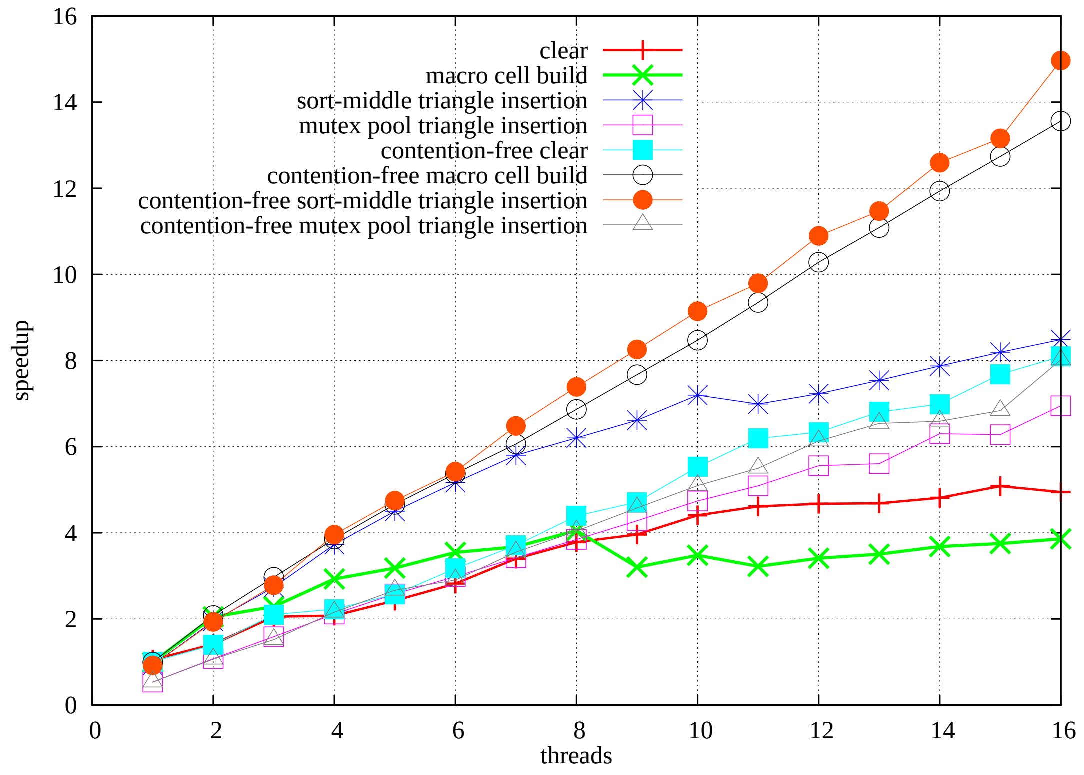


Figure 6.4. Simulation of ideal parallel computer. Graph of individual steps for the parallel rebuild, and then for the same parallel rebuild when each thread is forced to run by itself. Forcing the threads to run by themselves allows us to simulate build times when there are no resource contentions.

Thus, it is extremely important to make sure the algorithms make memory accesses as efficiently as possible.

6.4.3.1 NUMA. Our 8 dual-core processor computer is a NUMA system in which each processor also contains a memory controller. Normally, when a memory allocation is made, the OS assigns that memory to the node that first touches (reads or writes) the memory. Most grid implementations have a single thread allocate and clear the grid data in one operation, so most of the grid data will reside in a single node. Not only will one thread always have fast local access to the grid data, whereas the other threads have to remotely access the memory, but all the threads will have to share a single memory controller. If instead we interleave the memory allocations across the P nodes that need to access it, we can ensure that $\frac{1}{P}$th of the memory accesses will be to local node memory, and the memory bandwidth will be spread out among more memory controllers. Figure 6.5 shows how interleaving memory allocations across the nodes that need to access it, versus

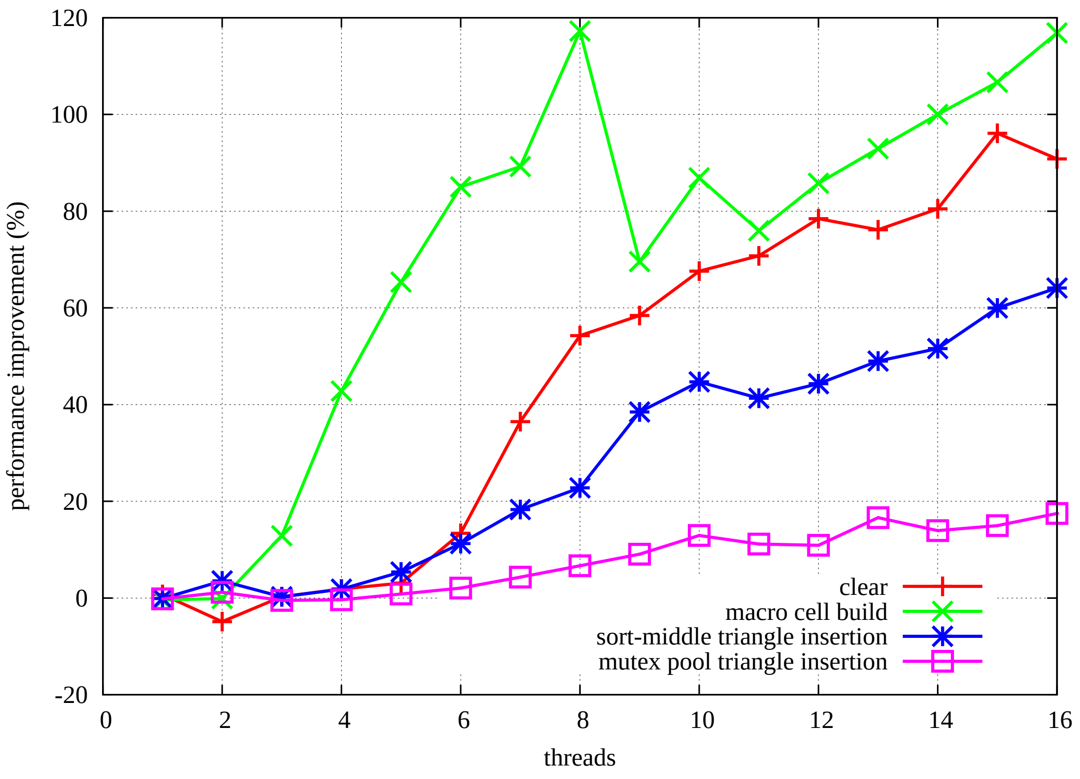


Figure 6.5. Interleaving memory. Comparison of interleaved memory allocations across the nodes to default OS supplied memory allocations.

allowing the OS to manage memory allocations, allows the build steps to scale to many more threads.

This clearly makes a large improvement at pushing back the memory bottleneck. The macrocell build step, for instance, consists of memory reads from many cells, memory writes to every macrocell, and very little computation. When we allow the OS to allocate the memory, the macrocell build becomes memory bound quite quickly, scaling to only two threads before hitting the memory bottleneck, at which point it cannot run faster despite adding more processors. From Figure 6.4 we saw that the macrocell build actually scales very well if only one thread runs at a time, so we know that this wall must be due entirely to a memory controller saturating its memory bandwidth. Sure enough, when we distribute the memory across all the nodes that might need to access it, the macrocell build time is able to scale up to 8 threads, at which point every node has a thread on it. Adding more threads does not add more memory controllers, limiting scalability due to a memory bottleneck beyond 8 threads.

The other build stages show a similar effect. In the case of the sort-middle build, not interleaving the memory causes a wall to be hit after 8 threads, preventing the sort-middle build from benefiting from more threads, and allowing the mutex pool build to catch up and surpass the sort-middle build. With memory interleaving, on the other hand, the sort-middle build is able to scale up to all 16 threads and outperform the mutex pool build. This implies that once the machine becomes memory bound, it might become advantageous to switch from the sort-middle build to the mutex pool build because it requires fewer memory accesses.

6.4.4 Comparison Using Other Scenes

The Thai statue is a shell of triangles that occupy a small number of grid cells, so we compare this scene to the 10 million triangle marbles scene depicted in Figure 6.6. This scene consists of 125,000 triangulated spheres randomly distributed inside a cube, uses a grid resolution of $216 \times 216 \times 216$, and macrocell resolution factor of 6. These two scenes share the same number of triangles, but the distribution of those triangles is very different, thus allowing us to see how robust the parallel grid rebuild is to triangle distribution.

Comparing the Thai statue with the marbles scene in Figure 6.7 shows the interesting property that the build times for the clearing and macrocell building for the two scenes are almost exactly swapped. For instance, the Thai macrocell build and the marbles cell clearing times are almost identical, even showing the same fluctuations. This is due to the fact that one scene has almost all the cells full whereas the other scene has almost none full. Furthermore, the cell clearing and macrocell building share the same basic iteration structure and level of computation, except that the clearing is optimized to skip cells when the macrocell is empty, and the macrocell build is optimized to skip cells when the cells are full. This implies that the total time spent clearing and building macrocells will not vary too much as long as the number of triangles and cells stays the same.

Both the Thai statue and marbles scenes consist of fairly uniform triangle sizes, so in Table 6.1 we also compare rebuild times with the Conference Room scene, which exhibits a large variation in triangle sizes, shapes, and locations, as seen in Figure 6.6. The Conference Room has 283k triangles, uses a grid resolution of $210 \times 138 \times 54$, and a macrocell resolution factor of 6.

Finally, to show that the parallel grid rebuild does not require large numbers of triangles in order to scale, in Table 6.1 we also compare against the 16K triangle hand model, which has a grid resolution of $72 \times 36 \times 36$ and a macrocell resolution factor of 6.

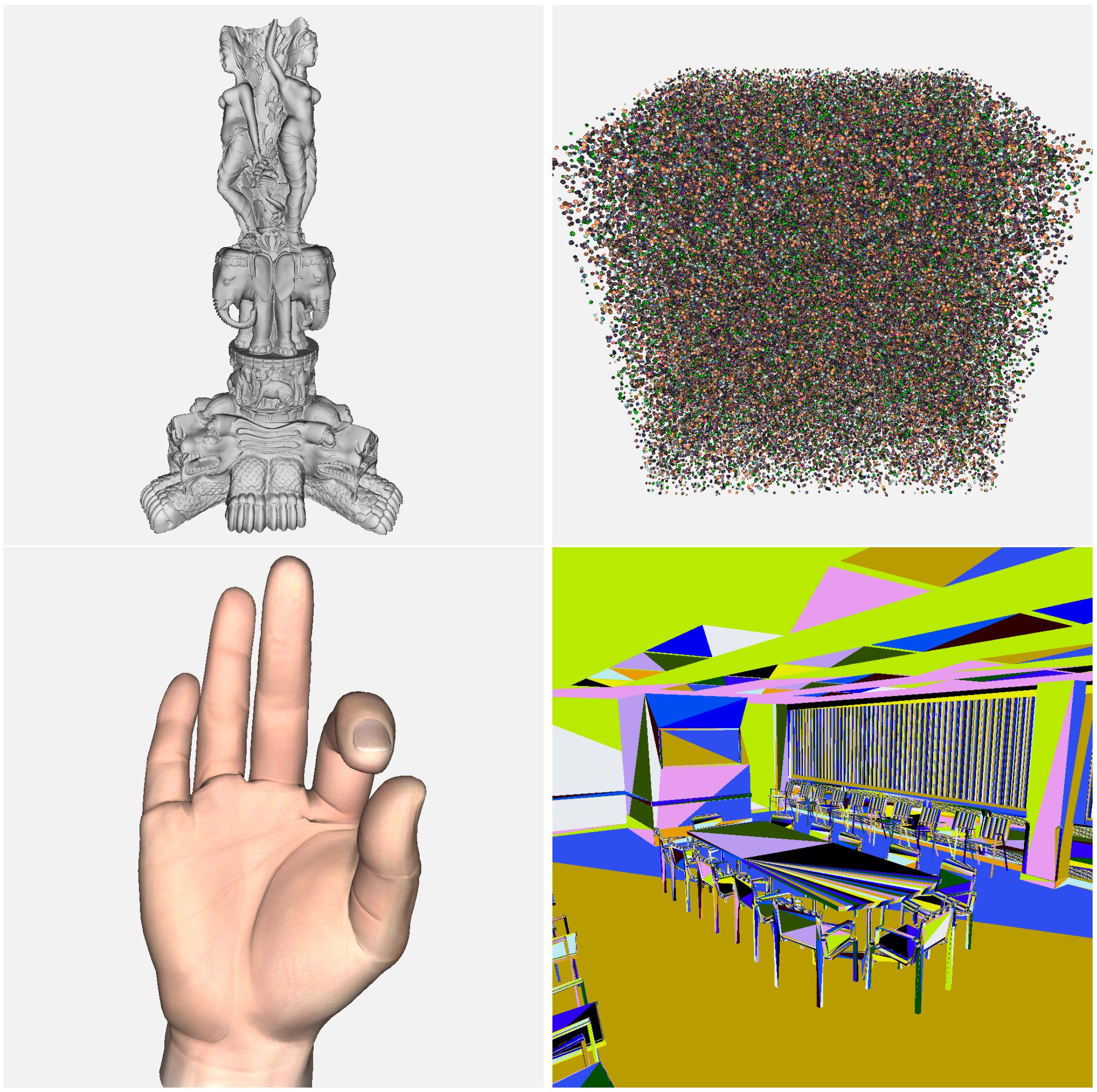


Figure 6.6. Test scenes. a) Thai Statue (10M triangles). b) 125,000 marbles randomly distributed inside a cube (10M triangles). c) Hand (16K triangles). d) The 282k triangle Conference Room exhibits a wide range of triangle shapes, sizes, and locations.

6.4.5 Overall Parallel Rendering Performance

Improving rebuild performance by parallelizing the rebuild is only worthwhile if it ends up improving the overall ray tracing performance. Since most animated scenes require rebuilding the grid every frame, we measure the total ray tracing time for a frame using three of the widely different scenes mentioned in the previous section.

As mentioned in [80], we also need to create the derived data for the triangle test described in [75]. Since the triangle rebuild shares some memory reads with the grid rebuild, we can do the triangle rebuild during the triangle insertion stage of the grid build.

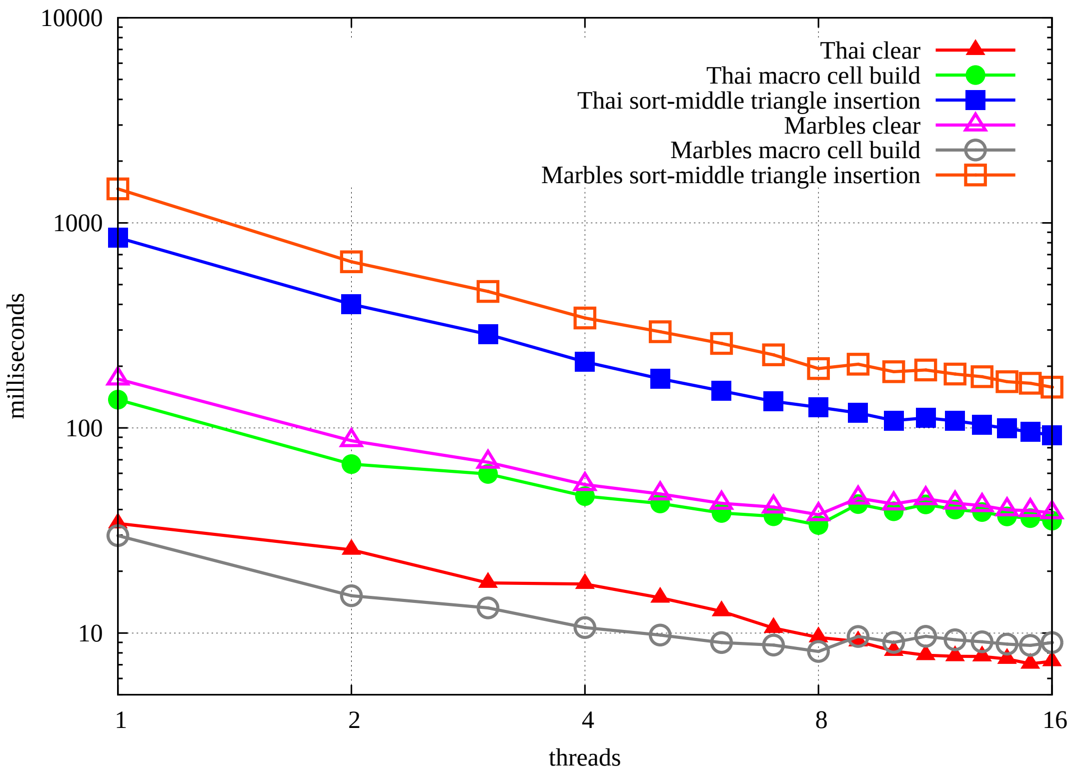


Figure 6.7. Effect of triangle distribution. Comparison between the 10M triangle Thai Statue and the 10M triangle Marbles scene for the sort-middle build.

Table 6.1. Rebuild scalability. Total rebuild times, in ms, for varying amounts of rebuild threads using the sort-middle build. 0 threads means serial rebuild.

scene	#tris	0	1	2	4	8	16
Thai	10M	955	1019	493	274	170	136
Marbles	10M	1548	1667	748	408	241	207
Conference	283K	89	93	49	34	26	21
Hand	16K	7.3	8.0	4.0	2.4	1.6	1.4

By doing so, not only do we remove a large amount of extra memory reads, we also space out the memory reads with the triangle build computations, which further reduce the memory pressure and allows the overall rebuild time to scale from 7x to 9x when using 16 threads. Even though including the triangle rebuild into the grid rebuild increases the overall performance and allows the grid rebuild to scale better, we chose not to include this in the other measurements since the triangle rebuild is not fundamentally required and other primitives might not require any rebuilding.

We measure the overall impact of the parallel grid rebuild by comparing the various parallel versions to the standard serial grid rebuild used in [80]; both triangle acceleration structure rebuild and rendering are parallelized in both versions, only the grid rebuild method is being varied. Figure 6.8 shows us how using the parallel grid rebuild allows the overall rendering performance to almost quadruple. For 16 threads, the Thai statue framerate went from 0.78fps with no parallel grid rebuild to 2.86fps with the parallel grid rebuild, the Marbles scene went from 0.50fps to 1.97fps, and the Hand from 78fps to 150fps. The Hand does not scale as well as the other scenes because it is so small; in fact, even without the grid rebuild (rendering only), it is only able to scale by 13x when using 16 threads.

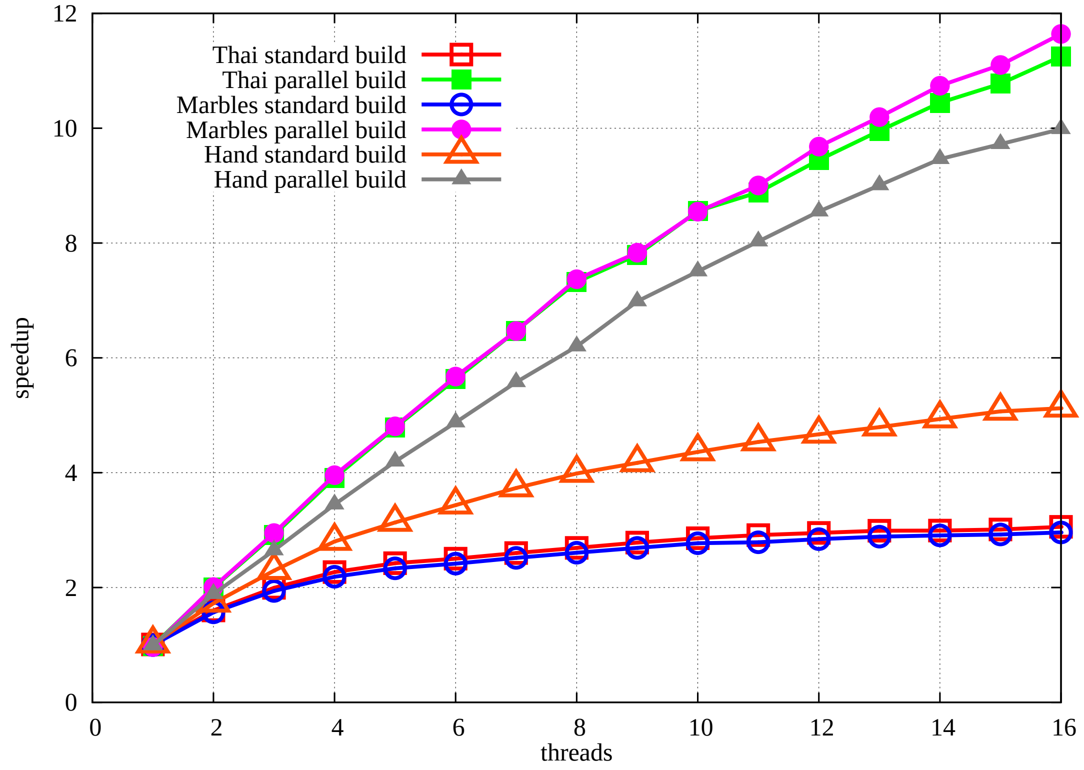


Figure 6.8. Overall scalability impact of the grid rebuild time on total rendering performance, on a 16-core machine. Data is given for both a serial grid rebuild, as well as for the sort-middle parallel construction technique evaluated in this chapter.

6.5 Summary and Discussion

We showed how the parallel rendering classifications used for 2D rasterization could be applied to parallel grid building and presented several parallel build methods based on these classifications. We showed that the sort-last approaches require significant thread synchronization, which results in these approaches incurring a large overhead, resulting in low parallel efficiency. These approaches almost double the build time. The sort-first approach is even less effective since it requires each thread to read all the triangles and compute which cells they overlap, making this approach expensive in both computations and memory bandwidth. The sort-middle approach was shown to have very little overhead over the serial build, which allowed it to perform better than the other approaches.

Memory bandwidth proves to be a major bottleneck in parallel grid rebuilds where the amount of computation performed is low and the amount of data that needs to be processed is high. Even the simple to parallelize grid cell clearing and macrocell building stages quickly started to show the memory pressure after just a couple threads were used. We were able to push back some of the memory pressure on our NUMA computer by interleaving the memory allocations across the nodes that access that memory. However, as we demonstrated with our "contention-free" parallel rebuilds, increasing the memory bandwidth would allow all our parallel rebuild methods to scale even better. Although the sort-middle build did perform best on our test computer, it is possible that for a computer with more limited memory bandwidth, the sort-last approach could perform better since it requires fewer memory accesses.

As we showed in Figure 6.8, using a parallel grid rebuild allowed the overall render speed to quadruple for the two large 10M triangle scenes and double for the very small 16K triangle scene. Thus, a parallel grid rebuild is important for any dynamic scene rendered with a grid when multiple CPUs are available. Since high-end consumer computers already ship with 8 cores, and there is a strong trend towards increasing number of cores, it is imperative that dynamic ray tracers be able to take advantage of the extra processor resources.

With memory bandwidth being the limiting factor, adding more cores to a chip might result in only minor performance gains if the memory bandwidth to the chip is not also increased. However, if we compare the Intel Nehalem processor with 8 cores and a memory bandwidth of 32GB/s to the dual core Opteron's bandwidth of 6.4GB/s, we see that for at least up to 8 cores, the memory bandwidth per core should at least stay roughly the same. Furthermore, since the memory for the 8 cores is all local to the processor, memory latency will be equally fast for each core, unlike with the NUMA-based system used for these tests.

As the number of computational cores in a computer continues increasing, a parallel grid rebuild algorithm will allow us to take advantage of these resources to speed up small to medium sized dynamic scenes, as well as providing scalable performance for large dynamic scenes.

CHAPTER 7

FAST, PARALLEL, AND ASYNCHRONOUS CONSTRUCTION OF BVHS FOR RAY TRACING ANIMATED SCENES

7.1 Introduction

Today, real-time ray tracing with dynamic scenes can be realized via either kd-trees, grids, or bounding volume hierarchies (BVHs), but there are trade-offs associated with each of these data structures [82]. Kd-trees seem to offer the highest ray tracing performance, but are most costly to build [79]; grids are efficient to build, but rely on a high degree of ray coherence that may not exist for complex scenes and/or secondary rays [80]. BVHs offer a compromise between performance and the ability to handle complex scenes and secondary rays, but are currently limited for many types of dynamic scenes. In particular, BVH-based interactive ray tracing systems are currently optimized for scenes that deform over time, and will deteriorate in performance for unstructured motion or severe deformations [78]. Although one can rebuild a deteriorated BVH every few frames to restore performance, this creates a disruptive pause while the BVH is being rebuilt [52].

In this chapter, we propose a new approach for handling dynamic scenes in a BVH-based ray tracer that is particularly targeted towards the highly parallel architectures we expect ray tracers to run on in the near future. In particular, we use Wald's fast binning-based SAH BVH build and his parallel variant of that build [76] and asynchronously rebuild the BVH concurrently with rendering and animation, thus allowing scalability to even larger models by allowing the (parallel) BVH build to run over multiple frames if required. This allows for (a) flexibly balancing the number of cores allocated to rendering and rebuilding; (b) quickly rebuilding from scratch if required; and (c) rendering scenes faster than could be done by from-scratch rebuilding alone, especially for larger scenes.

7.2 Background

BVHs are both fast to traverse and fast to update when the scene deforms (by refitting the BVH bounds) [78]. For this reason, BVHs have recently become a popular acceleration structure in interactive ray tracers.

7.2.1 Handling BVH Deterioration

Although refitting a BVH is inexpensive, it does have several drawbacks. First, it is only applicable for deformable scenes (i.e., scenes that do not change the triangle count or vertex connectivity). Second, refitting a BVH will result in a *correct* BVH, but it will not necessarily be *efficient*. The refitted BVH retains the original frame's BVH topology, but as the scene deforms, the triangles might form a configuration for which a different structure might yield better performance. This will eventually lead to a deterioration of BVH quality (and performance) as scene and BVH become out of sync.

As pointed out by Lauterbach et al. [52], deforming a BVH usually works for at least some number of frames, and instead of rebuilding a BVH every frame, one could rebuild only every few frames, with the frames in-between handled by BVH deformations. In order to do these rebuilds when they are most effective, Lauterbach et al. [52] have proposed a "rebuild heuristic" that detects BVH degradation, and rebuilds the BVH if and only if the quality degradation has reached a given threshold. This allows for striking a balance between total rebuild cost and render cost, and can yield a significantly reduced average frame time in an animation.

Unfortunately, a lower *average* frame time is not always helpful in an interactive setting. In an offline animation the infrequent rebuilds can be amortized over all frames of the animation, yielding a low average time per frame; in an interactive setting, however, amortization does not apply, and system responsiveness is disrupted while a rebuild is performed, which hurts the user's ability to interact with the environment.

In order to distribute the rebuild cost more evenly over multiple frames, Yoon et al. [88] have proposed to selectively restructure an existing BVH on a frame by frame basis—thereby incrementally fixing some of the accumulated deterioration. For every frame, they first deform the existing BVH, then analyze this BVH to detect nodes of high overlap (i.e., deterioration), and selectively restructure pairs of nodes to reduce this overlap. Although very effective at avoiding deterioration, the method also carries a certain cost, may be nontrivial to parallelize, and has not yet been demonstrated in a real-time setting.

7.2.2 Fast Construction of SAH Hierarchies

Instead of only partially or infrequently rebuilding a BVH, the simplest and most robust method for handling dynamic motion would be to rebuild every frame. This is in fact the biggest advantage of grid data structures, which can be easily rebuilt per frame [41].

For kd-trees, fast "scan-based" approaches to building SAH-based data structures have been proposed independently by Popov et al. [64] and Hunt et al. [37]. Though these two publications report rebuild rates of roughly 150,000 and 300,000 triangles per second, respectively, this is still insufficient for achieving truly interactive frame rates for anything but rather small models. More recently, Shevtsov et al. [69] have shown that these techniques can also be parallelized in a scalable fashion, and that near-real time rates can be achieved on multicore processors. If some information from the scene hierarchy is available, lazy building from the hierarchy can be up to an order of magnitude faster than building from a triangle soup [36].

For BVHs, fast rebuilding has received far less attention. The fastest known (single-threaded) build method is the BVH variant of the fast spatial median build Wächter et al. [74] have proposed for their bounding interval hierarchy (BIH). Though subtly different from a standard median build [82], this BIH build essentially performs spatial median splits, and thus achieves lower traversal performance than SAH-based builders.

More recently, Wald showed that fast scan-based kd-tree build techniques can also be applied to BVHs, and that they work at least as well for BVHs as they do for kd-trees [76]. Using these techniques on a dual 2.6 GHz Clovertown PC, single-threaded rebuild performance of up to 1–2 million triangles per second have been achieved. Using parallelization, this rate could be raised to up to 7 million triangles per second, at which rate the system on that particular hardware stopped scaling, apparently due to bandwidth limitations. Since we use that algorithm in our system, we will describe it in more detail below.

7.3 System Overview

Summarizing previous work, we can conclude that BVHs hold promise for ray tracing dynamic scenes, but that each of the individual techniques proposed for handling animations has limitations: refitting is fastest, but eventually leads to degraded performance if deformations become too severe; infrequent rebuilding amortizes rebuild cost but leads to disruptions in frame rate; fast builds reduce the build time but are not fast enough on their own; and finally, parallel binning is even faster, but does not scale once the bandwidth wall is hit, and still limits frame rate to how fast a full BVH can be built.

Some of these problems will likely become exacerbated by future ray tracing systems running on architectures with a large number of cores. However, by building the BVH asynchronously to the rendering and refitting, we can get the advantages of these techniques while minimizing their respective disadvantages. In particular, we can combine the fast and parallel BVH building [76] with asynchronous rebuilding [39], which leads to three distinct advantages:

- Using a fast parallel builder with the asynchronous rebuilding significantly reduces the time the asynchronous builder has to wait for a new BVH, and thus reduces the potential for deterioration. At the same time, it also provides the system with a high quality SAH-based BVH instead of lower quality but faster to build non-SAH builder.
- For scenes where rebuilding is still too slow for per-frame rebuilds, the asynchronous approach allows for amortizing the build time over multiple frames.
- Even though the parallel builder does not scale to all CPUs, building asynchronously allows for using all of the system's cores by assigning as many cores as reasonable to rebuilding, and all other ones to rendering.

7.4 Fast Construction of SAH-based BVHs

In any ray tracer, there is a trade-off between build quality and build time. Better data structures achieve higher performance, but if building them takes too long, the total frame time may still go up. Although the asynchronous approach somewhat decouples build time from render performance, the basic problem still exists. If the build takes too long, too much deterioration can occur before a new BVH is built, resulting in a slower and choppy frame rate.

Consequently, our original paper on building the BVH asynchronously [39] recommended a spatial median BVH instead of a better SAH BVH. Though yielding somewhat better BVHs, the high-quality SAH build took about $20\times$ longer to build, which eventually led to too much deterioration. Since then, however, Wald [76] has shown that high-quality BVHs can also be built much faster when using the same techniques as recently proposed for fast kd-tree construction [37, 64]. Since these techniques are nearly as fast as spatial median splits and nearly as good as SAH builds, we have decided to adopt them in our system.

In particular, these techniques do not evaluate all possible kd-tree split planes, but use a given set of sample planes. The triangles are then projected into the "bins" formed by these planes, and the number of triangles in each bin are recorded. The algorithm then evaluates the SAH for each plane, and selects the one with lowest cost. Though approximate in nature, for kd-trees, it has been shown that as few as 8 planes usually suffice to be within a few percent of the optimum [37].

Though originally designed for kd-trees, the same techniques apply to BVHs as well. The only difference is that in the kd-tree case, triangles overlapping the plane have to be counted in both halves, and the plane determines the exact bounds of each subtree. In a BVH, a triangle can be on only one side of the plane, and consequently, the subtree bounds can overlap the split plane. Therefore, we track not only the number of triangles in each bin, but also the bounds of all triangles projecting to this bin. Based on these bins, we can then compute the number of triangles as well as their spatial extent to the left and right of each sample plane, compute the SAH, and select the best plane.

During the build, we never need a triangle's exact shape, but only its axis-aligned bounding box (AABB), as well as one representative point for binning (for which we choose the AABB's centroid). To make use of SSE SIMD extensions, we store centroids and AABBs in SSE 4-float format, and precompute those in the beginning. During the build, each binning operation is then only a few instructions long.

As investigated in more detail in [76], using this method we can build a SAH-based BVH roughly 10$\times$ faster than the sweep-based SAH build outlined in [78]. Approximate high-quality BVHs can be built with as few as 4 bins, and 16 bins are usually close to the optimum (see Table 7.1). At 16 bins, the near-optimal SAH build is only about 2$\times$ slower to build than a spatial median build, while still producing high-quality BVHs.

7.5 Parallel Construction of SAH-based BVHs

There has been a recent explosion in the number of processor cores available on consumer computers, with high-end PCs already containing 8 or more CPU cores. We argue that this trend is likely to continue and that soon, architectures with dozens of cores will be commonplace. With that in mind, designing algorithms in a parallel way will soon be critical, in particular for real-time applications like ray tracing.

Table 7.1. Build and render performance of various builds. Single thread absolute build times and relative render performance to SAH sweep for an SAH sweep build, median-split BIH-style build, and fast binned SAH build, for the animated scenes described in Section 7.7. Build times are in milliseconds, on a 3.0 GHz Opteron CPU (one thread).

scene	Fairy Forest 2	BART	Expl. Dragon
sweep	3006ms (100%)	2270ms (100%)	1614ms (100%)
BIH	216ms (72%)	104ms (75%)	70ms (87%)
binned	364ms (94%)	347ms (95%)	185ms (104%)

The obvious way of parallelizing an algorithm producing a binary tree is to have different threads work on different subtrees. Though simple and effective, this technique requires a certain number of subtrees to work on. One way of doing that is to start with a single thread working on the root node and then adding another thread after every split. Although simple to implement, not all threads are active from the beginning, and since the first splits unfortunately are the most costly ones, this technique does not scale well. Wald therefore proposed a two-stage approach: in the first stage, all threads collaboratively generate a coarse partitioning; the second stage then switches to different threads that work on different subtrees [76].

Wald showed two ways for implementing the first stage. The first method involves parallel grid-based splitting whereby the bounding box of the triangles' bounding box centroids are uniformly subdivided into a uniform grid. Each of the T threads create their own copy of that grid and take one T^{th} of the triangles. Then in a single pass over those triangles, each thread bins each triangle into the cell that its centroid projects to. Once all the threads finish, the T grids are merged in parallel (similar to the parallel grid build of Ize et al. [41]). Once merged, one of the threads builds the BVH nodes that represent that grid, which is very similar to Wächter's BIH technique.

The other method uses parallel binning. In this case the T threads collaborate on each individual partition. First, each of the T threads compute a bin histogram as described in the previous section, albeit only for one T^{th} of the triangles. Once done, one thread merges these histograms, selects the best split, and creates a new BVH node. Based on the selected split, each of the threads then splits its part of the triangles into a left and right half, and copies the two halves into the respective regions of the triangle ID array (which fortunately can be predetermined by the known sizes of all threads' bins).

Though originally designed for parallel SAH construction, the same framework can also be used for parallel spatial-median building. In particular, the grid-based splitting essentially constructs a spatial median build in the coarse partitioning anyway, so all that needs to be done to have a parallel spatial median BIH builder is to use the existing single-threaded BIH builder in the parallel subtree phase. This way, we eventually have three different parallel build schemes: a fully SAH built tree using a parallel SAH binning on the top with binning for each subtree, grid partitioning with binning for the subtrees results in a spatial median at the top and SAH for each subtree, and grid partitioning with a BIH build per subtree results in a spatial median throughout.

7.5.1 Results

In Table 7.2, we give build performance and relative build performance for these two parallel build methods. Due to space considerations, we only report data for 2, 4, and 8 threads, as on the particular hardware we use (an 8-way dual-processor Opteron) the parallel build stops scaling after 8 threads anyway (probably due to being bandwidth-limited). As can be seen by this table, a parallel build scales extremely well to 2 threads, and continues scaling, but with worsening efficiency, up to 8 threads. In particular, we can build a fully SAH-based BVH faster than a single-threaded BIH-build. Still, when properly parallelizing the BIH build as well, for the same number of threads a BIH build is still about twice as fast as a SAH build, so the original trade-off between build time and BVH quality remains at least to a certain degree (the ratio in build times dropped from 20:1 to 2:1, but did not entirely disappear).

Table 7.2. Build scalability. Build times when building on 2, 4, and 8 threads. Grid+BIH is spatial median throughout, SAH+SAH uses a SAH for both coarse partitioning and for each subtree. Numbers in parenthesis denote speedups over single-threaded building as given in Table 7.1.

scene	#threads	Fairy Forest 2	BART	Expl. Dragon
grid+BIH	2	75ms (2.9×)	41ms (2.5×)	29ms (2.4×)
SAH+SAH	2	196ms (1.9×)	154ms (2.3×)	97ms (1.9×)
grid+BIH	4	61ms (3.5×)	27ms (3.9×)	19ms (3.7×)
SAH+SAH	4	120ms (3.0×)	95ms (3.7×)	58ms (3.2×)
grid+BIH	8	53ms (4.1×)	19ms (5.5×)	14ms (5.0×)
SAH+SAH	8	84ms (4.3×)	58ms (6.0×)	45ms (4.1×)

7.6 Asynchronous Dynamic BVHs

Even with a fast and parallel BVH builder, two problems remain. First, due to bandwidth limitations, the parallel builder does not scale beyond a certain number of CPUs, leaving all other CPUs idle. Second, the absolute build time still places an upper limit on the frame rate that can be achieved.

Therefore, we propose to never wait for rebuilds, and instead perform all rebuilds *asynchronously* to normal rendering; while a new BVH is built on an application-specified number of K rebuild threads, the remaining $N-K$ threads proceed with rendering by deforming the most recently finished one. As soon as a new BVH is available, it replaces the currently used one. On a machine with N cores, this results in K dedicated rebuild threads, and $N-K$ dedicated update/render threads (see Figure 7.1). The actual number of rebuild threads currently has to be specified by the user upon startup; changing the number of rebuild threads dynamically during runtime (ideally in an automated way) provides for some interesting avenue of future work, but is not currently supported.

To allow multiple threads to work asynchronously on the same data, we have to double buffer the shared data, which consists of the vertex positions and, of course, the BVH nodes. All other data, like triangle connectivity, triangle acceleration structures, vertex normals, texture coordinates, etc. are not touched by the builder, and so are not replicated. This results in roughly 80 bytes extra storage per triangle, which for most scenes, has a minor impact. A large 1M triangle scene, for instance, would only require roughly 80MB of extra storage.

Whenever a new BVH is finished, the (parallel) rebuilder passes it to the $N-K$ rendering threads, and grabs a new set of vertices to work on. This naturally occurs between when the render threads finish their current frame and before they start refitting the BVH for the next frame. Since at that time, the application has not yet computed the new vertex positions, we start the build process with vertex positions that are already one frame outdated. Although we could wait for the new vertex positions to be calculated before exchanging the data, this would require an expensive copy of those values to the rebuilder, which is especially problematic since the render threads must be blocked waiting for the copy to finish before they can use the new data. This critical section would hurt the system's scalability. Instead, by building the BVH from the last finished frame's vertex positions, the vertex and BVH buffers can be switched quickly with two pointer swaps, and the builder, application, and render threads can immediately continue. Furthermore,

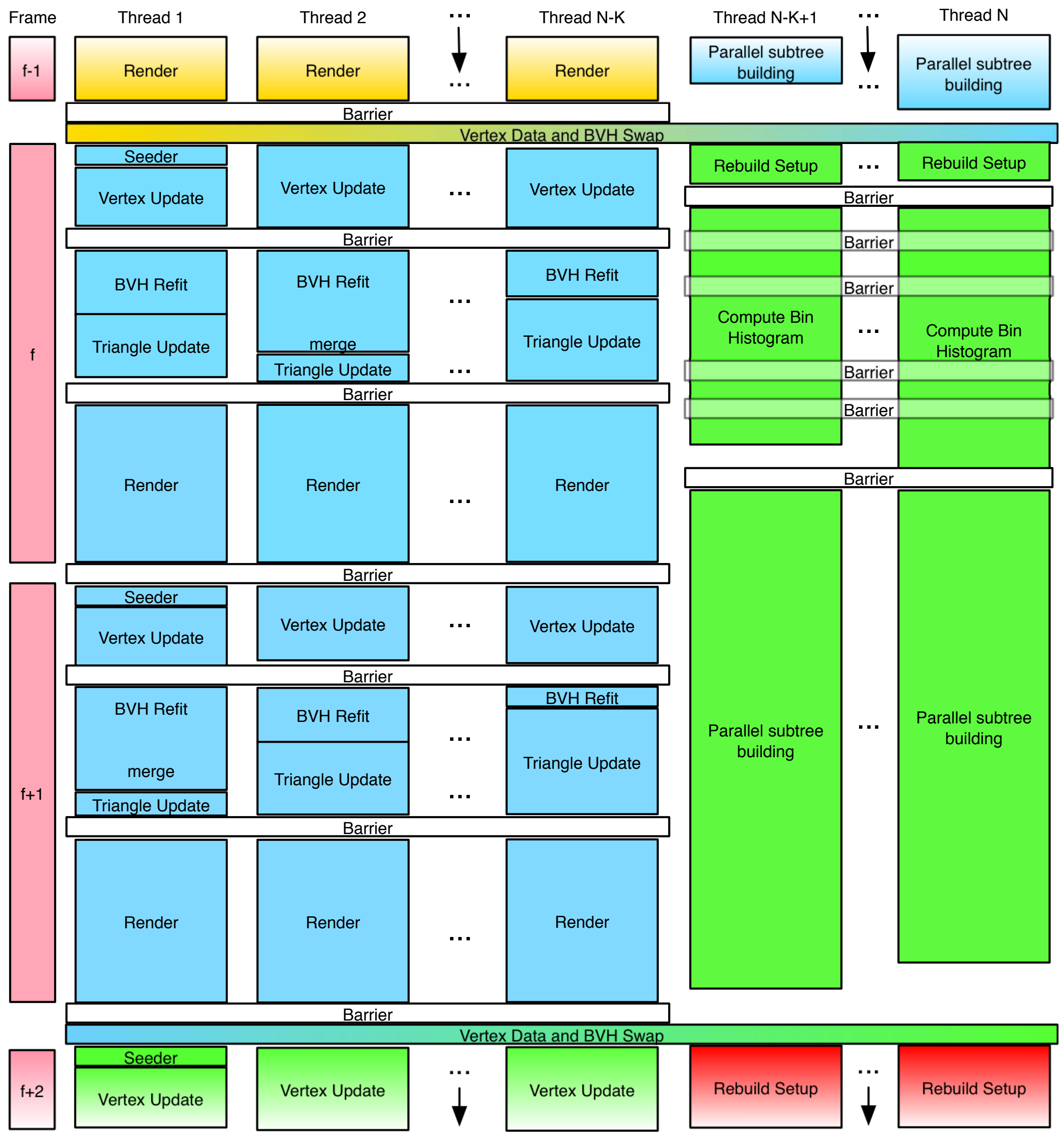


Figure 7.1. Asynchronous framework. Given a highly parallel architecture with N cores, $N - K$ of the cores work on parallel rendering and BVH refitting of the most recently finished BVH, while the K other cores work asynchronously and build in parallel the new BVHs as fast as possible, potentially over multiple frames (2 in this example). BVHs are deformed for only a few frames, and both scalability bottlenecks and pauses are avoided altogether.

since it will likely take several frames before the new BVH is available anyway, that BVH will already be outdated by the time it is finished, so building the BVH with vertex positions that are outdated by one frame is equivalent to the build taking one frame longer to complete—which is a minor cost for ensuring good scalability.

7.6.1 Parallel Rendering and Update Steps

With the BVH build decoupled from all other per-frame computations, the rendering stage itself can be kept highly parallel. In particular, the following operations are being performed by the render and update threads.

7.6.1.1 Vertex generation. Vertices are usually generated by the application using, for example, a vertex shader or linear interpolation. Since even generating the vertices can be costly compared to refitting a BVH or rendering a frame, ensuring good system scalability requires parallelizing the vertex generation, too. In our current framework, we compute vertices by linearly interpolating between fixed timesteps, which we do in parallel on the $N - K$ update/render threads.

7.6.1.2 Parallel BVH refit. Once the new vertex positions are known, we refit the most recently finished BVH's bounding volumes in parallel. To ensure scalability, we use a three-way dynamic load-balancing scheme for the update. In the first phase, one "seeder" thread traverses the upper k levels of the BVH and records the node IDs of all the leaf nodes encountered and the node IDs of the k'th level subtrees. Though no other thread can start refitting until this seeding is done, there is no scalability issue as the seeding is extremely fast.

Once all the $N - K$ render/update threads are done updating the vertices and seeding, they synchronize on a barrier, and then switch to BVH refitting. We dynamically load-balance by having each thread take a node ID from the list, refit that subtree, and repeat. As soon as a thread finds no more subtrees to refit, it immediately goes on to performing triangle updates. The last thread to *finish* a subtree update also performs the final "merge" of the refit subtrees.

7.6.1.3 Triangle update. For ray-triangle intersection, we use the method outlined in [84]. This method uses a precomputed set of data values for each triangle, which for an animated scene has to be recomputed every frame. Due to imbalances in the BVH refitting phase, the update threads can enter that phase at different times. We compensate by dynamically load balancing the triangle updates. In this way, all of the individual operations—parallel subtree update, serial subtree merge, and triangle update—are fully interleaved, ensuring that all rendering threads remain constantly utilized and finish at the same time.

7.6.1.4 Parallel rendering. Once all update/render threads have finished updating, they synchronize themselves via a barrier, and then render the scene using a standard tile-based dynamic load balancing scheme. All per-frame operations—update and render—are dynamically load-balanced at all stages, and only three barrier operations are performed per frame: after vertex updates, after all threads have completed updating, and once all tiles have been rendered. The only nonparallelizable stage is the time between the end of the current and the start of the next frame, in which the application processes user input, displays the image, and if applicable, swaps the rebuild data. Even then, parallel rebuilding is active in the background.

7.6.1.5 BVH build method. The choice of BVH build method—as well as the number K of threads used for rebuilding—is completely orthogonal to the asynchronous rendering approach. When building asynchronously, a BVH will *always* be outdated by as many frames as it took to build this BVH. Thus, there is still a trade-off between a build method's resulting BVH quality and the time to achieve that quality, as longer builds potentially suffer worse from deterioration. Similarly, there is a trade-off related to how many threads are used for rebuilding, with more threads reducing the amount of BVH deterioration, but also takes resources away from rendering. These trade-offs we will investigate in more detail below.

7.7 Results and Discussion

Although mainly designed for upcoming multicore architectures that will likely contain a large number of cores, current processor architectures only feature 2 to at most 4 cores per processor. We therefore use an 8 processor dual-core Opteron 8222SE shared-memory PC for our experiments, as that gives us a total of 16 cores, which we believe more closely resembles the number of cores on future hardware. Unless otherwise noted, we use all 16 cores in our tests. We use three test scenes: the "Fairy Forest 2," the "Exploding Dragon," and the "2×2 BART museum" (see Figure 7.2). The Fairy Forest 2 is a 7.75s long keyframed animation with 394K triangles, almost all of which are slowly deforming every frame, and resembles a game-like scene. The 252K triangle Exploding Dragon scene is 3.2s long and exhibits complex deformations occurring very quickly. The 262K triangle 2×2 BART museum is 8s long and composed of 4 copies of the museum scene from

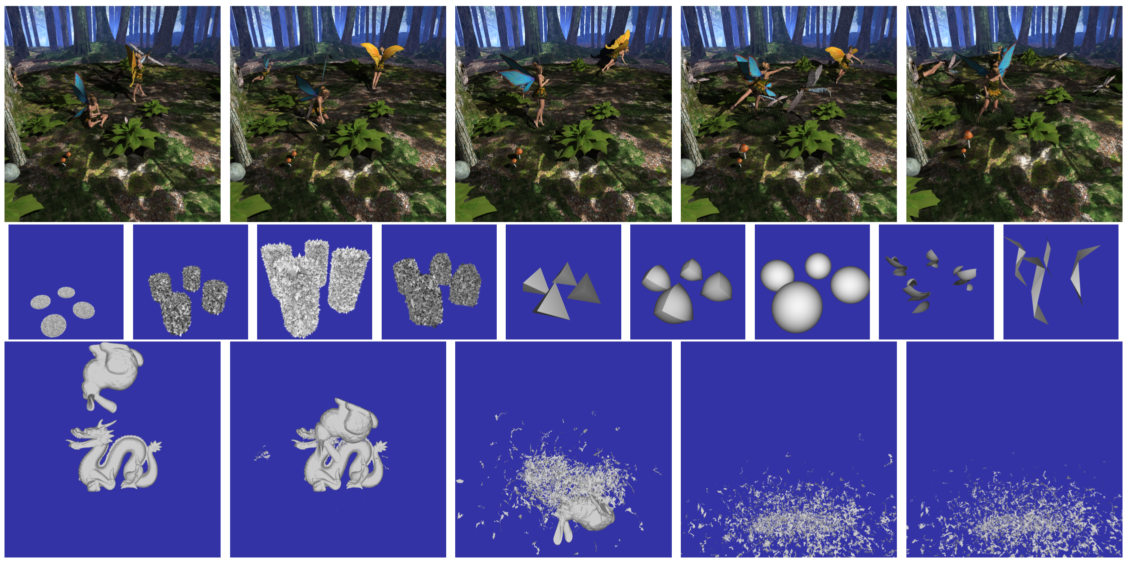


Figure 7.2. Test scenes. The three scenes used for evaluating how our system performs for various kinds of motion. Top: “Fairy Forest 2” with 394K triangles. Middle: “BART 4” (part of the BART benchmark replicated 2x2 times), showing intentionally incoherent motion of 262K triangles. Bottom: The UNC “Exploding Dragon” of 252K triangles.

the Benchmark for Animated Ray Tracing (BART) [54] and is intentionally designed to stress test large deformations. Because the BART scene deforms heavily by morphing into wildly varying shapes (see Figure 7.2), it provides a challenge in which standard BVH refitting quickly breaks down (see [52, 78]). Finally, the UNC Exploding Dragon also shows incoherent motion, but in a less artificial setting.

All measurements were performed using the packet/frustum ray tracer used in [78]. To simulate the effect of a somewhat higher render-to-update cost ratio, as would occur with more complex shading effects, the Fairy Forest 2 is rendered at 2048×2048 with lambertian shading and hard shadows. The other two scenes are rendered at 1024×1024 pixels with no shading.

7.7.1 Comparison to Synchronous Approaches

The first obvious question to investigate is how our asynchronous approach compares to traditional synchronous approaches. In Figure 7.3, we compare our method to the three standard approaches of (a) not rebuilding at all (“refit only,” as used in, e.g., [78]), (b) rebuilding every frame as fast as possible (“full rebuild,” [76]), and (c) rebuilding infrequently as indicated by the “rebuild heuristic” as proposed in [52]. For both the full rebuild and the rebuild heuristic, we use the fast, parallel BVH build described before.

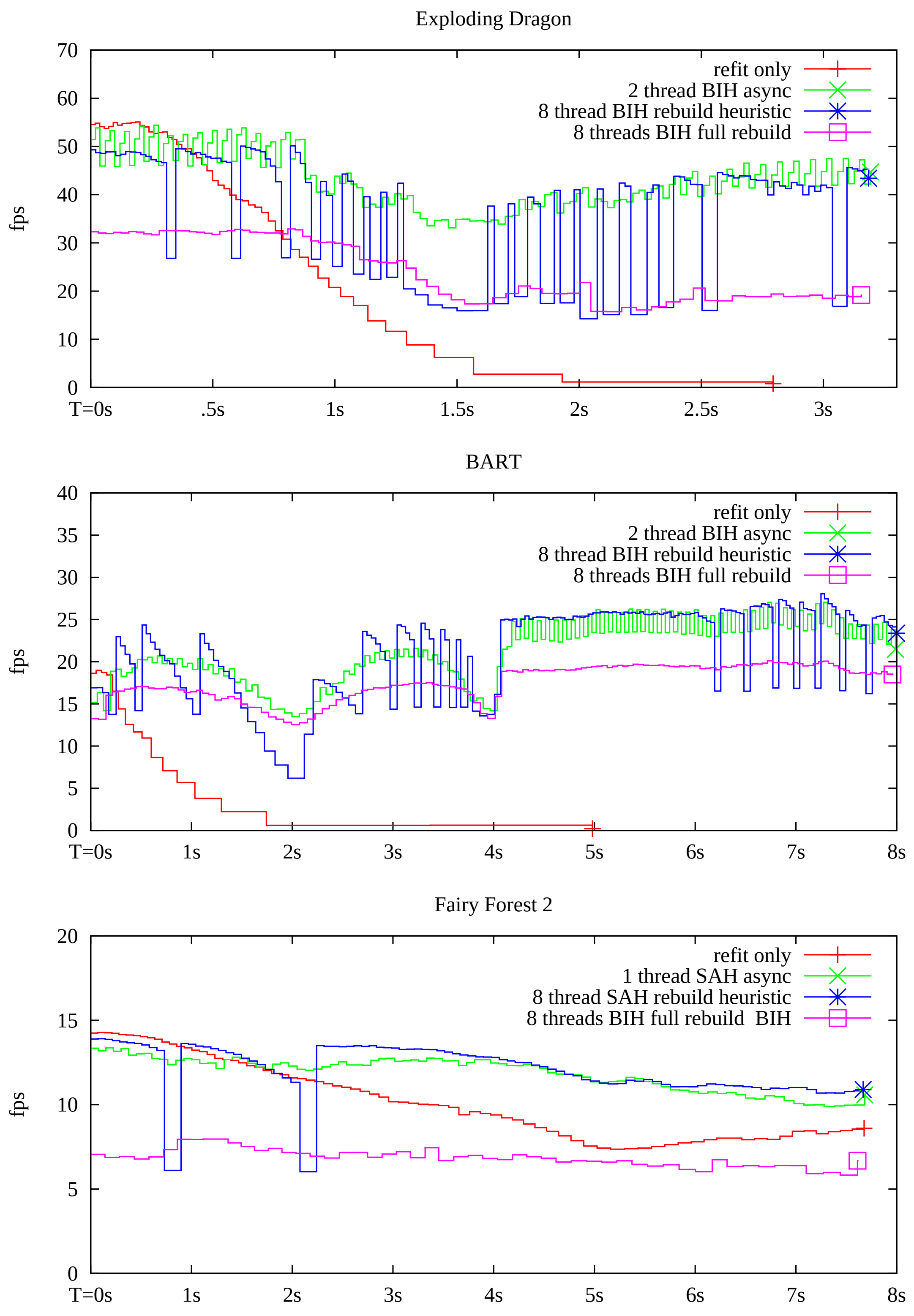


Figure 7.3. Impact on frame rate using various build strategies. For the BART and Exploding Dragon scenes, 2 thread BIH performed best for the asynchronous build, whereas 1 thread SAH performed best for the Fairy Forest 2 scene.

We found that using 8 instead of all 16 cores gave the fastest rebuild times, and so use 8 threads for these (synchronous) rebuilds.

As can be seen from Figure 7.3, all scenes show that simply refitting leads to severe performance deterioration, with about a 1.5× drop for the Fairy scene, and a nearly complete standstill for the BART and Dragon scenes. Lauterbach's approach is clearly superior to refitting only; it avoids the BART scene's extreme deterioration, and achieves higher frame rates for the Fairy scene. Furthermore, with only two rebuilds triggered for the Fairy scene, it achieves nearly optimal frame rates for most of the animation.

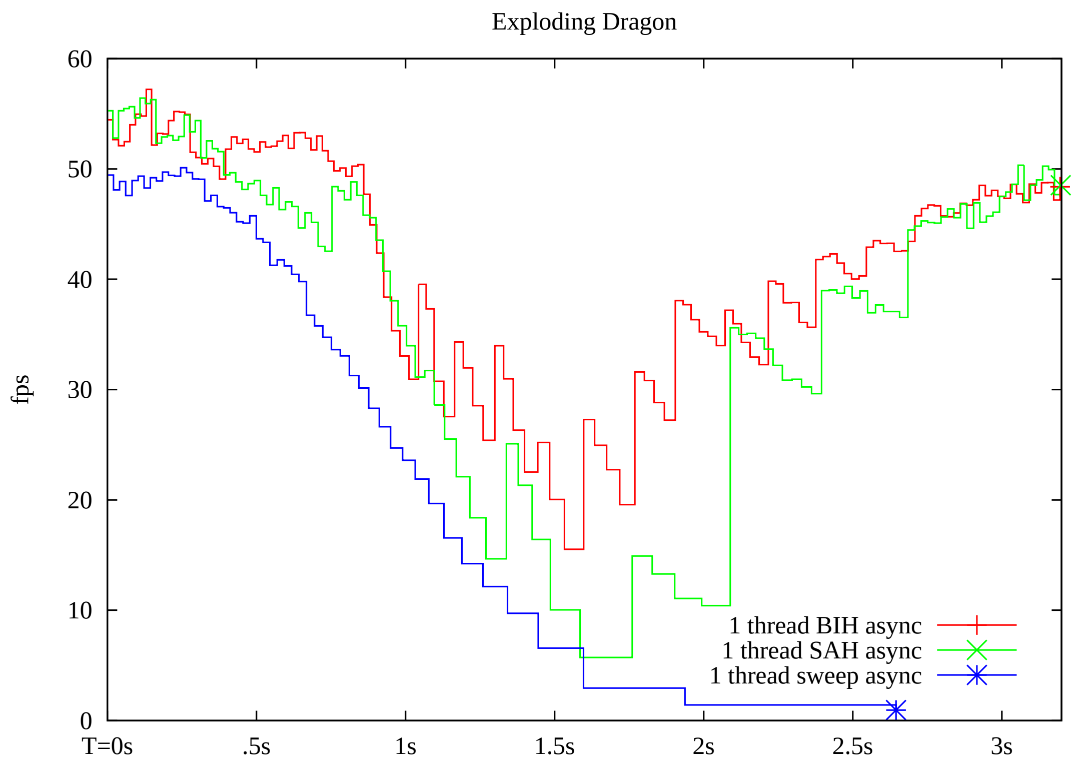
Although these experiments confirm Lauterbach's rebuild heuristic is superior over deforming only, they also show its weaknesses: the high variation in frame rate caused by rebuilds and varying rates of deterioration, the "sawtooth" effect of deterioration until a new build is triggered, and, in particular, the disruptions in which the system temporarily freezes when a new BVH is being built. Per-frame rebuilding avoids this effect, and produces the smoothest frame rate. However, it usually exhibits the lowest frame rate of all the methods, even when parallelized.

Compared to these methods, our method is clearly advantageous. Though the rebuild heuristic or refitting only can reach higher peak performance (because less cores are used for rendering), our method is clearly competitive where these techniques are best, and clearly outperforms per-frame rebuilding. In addition, our technique suffers from neither accumulated deterioration (as in refitting) nor from severe sawtooth effects and disruptions (as with the rebuild heuristic).

7.7.2 Fast vs. Slow BVH Building

As mentioned above, asynchronous rebuilding can decouple build time from render time, but the longer the build takes, the more BVH deterioration can accumulate in the currently refitted BVH. Obviously this effect depends on how severely the scene deforms (i.e., on the speed of the animation and the kind of motion). To quantify this effect, we have measured the frame rate for both the original sweep SAH build, the fast binned build, and a BIH build (each using one thread).

The results of this comparison are given in Figure 7.4. Though the sweep build should produce the best BVH quality, it consistently achieves much lower frame rate than the BIH build or the fast binned build, as too much deterioration has accumulated by the time the BVH is built. Note that this is true even for the first frame rendered with a newly available BVH (around $t = 6s$), as that newly available BVH is already outdated.



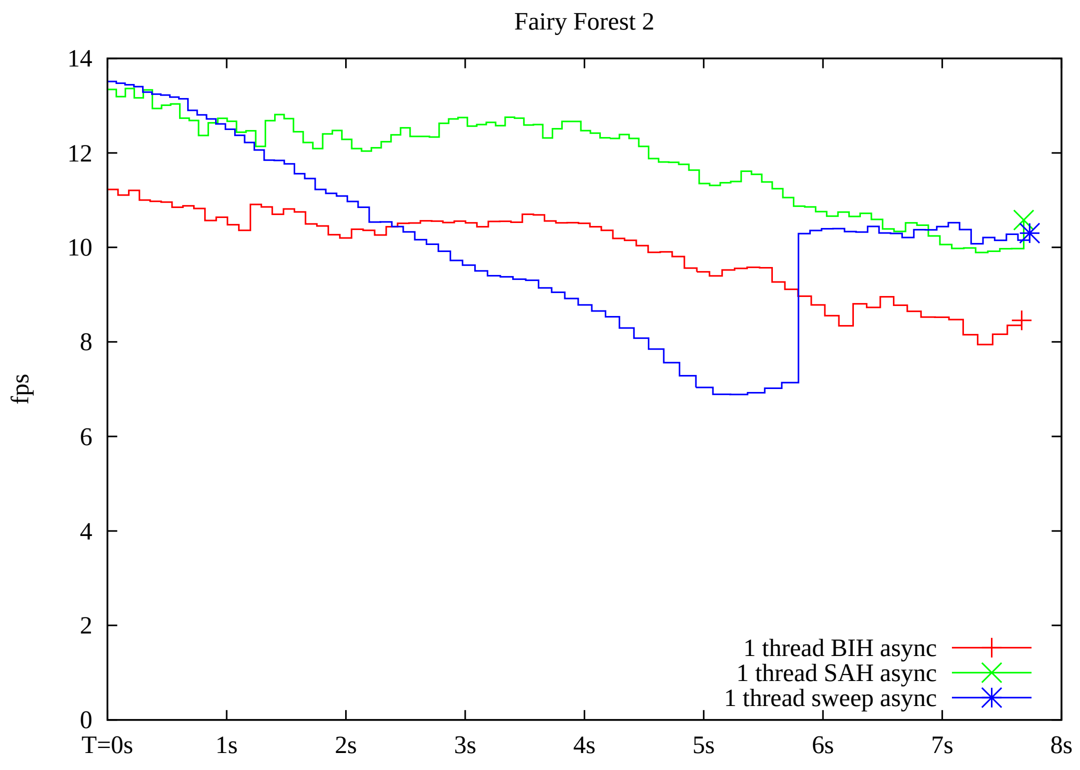


Figure 7.4. Impact of build time in the asynchronous system for single thread BIH, binning, and sweep SAH. For the Fairy with high render-to-rebuild cost and smooth deformation, the SAH build wins, while for the Dragon scene with very simple shading, the BIH build performs at least as good. Both BIH and fast SAH are clearly superior to the sweep build.

If deformation is as severe as in the Dragon or BART scenes, slow building results in a severe sawtooth effect, or even a complete breakdown in performance.

7.7.3 Relation on Build Time and Frame Rate

It is important to realize, however, that as long as the animation is specified in world time (as is usually the case), this sawtooth effect is *not* related to render performance at all, but only to how much world time has to be bridged with deformation. Higher or lower render performance via different hardware or shaders, for example, would change the number of frames rendered during the time needed to build the new BVH, but the amount of deterioration accumulated in that time (and thus, the relative performance impact) would not change.

7.7.4 Number of Threads and Choice of Build Method

Section 7.7.2, showed that reducing the build time can be crucial for avoiding the accumulation of deterioration. On the other hand, Section 7.7.3 demonstrated that it is only the *absolute* build time that matters, not its relation to render time—which suggests that once a certain rebuild performance has been reached, either through more cores, faster cores, or better build algorithms, building even faster will have a diminishing return.

Thus, the exact balance on how many threads and which build method to use cannot be answered without knowing what the actual hardware and scene characteristics are. Generally, for scenes with slow smooth deformations, a single build thread may suffice, whereas for scenes with many triangles or with severe deformations, using more threads for rebuilding is advantageous even if that takes resources away from rendering.

Similarly, the question on whether to use the BIH build or an SAH build remains hard to answer. Even the fast SAH build takes around twice as long as a BIH build, so for the same absolute build time using a BIH build would free more threads for rendering. For the same number of build threads, the BIH build would finish sooner, and the reduced amount of accumulated deterioration can easily offset the lower quality trees in rapidly deforming scenes. Furthermore, the BIH build does not subdivide nodes as much as an SAH build, and whereas this does lead to fatter leaves and slower rendering, it also makes both building and refitting faster. For simple scenes, the refitting time can be equivalent to the rendering time (the Exploding Dragon scene takes 11ms to refit and 6–13ms for rendering), so the improved refitting time can easily compensate for the slower rendering time. Note that the parameters of an SAH build can also be modified to produce shallower trees.

Thus, it is impossible to come up with a configuration that is the best for all scenes and hardware configurations. For our current 16-core system, we usually use one or two threads for rebuilding, and would likely use up to 4 build threads if the scenes would get larger and the deformations more severe. As to build method, we usually use the fast SAH for low deforming scenes (usually the common case), and the BIH build when the rate of deformation is high. With this configuration, our system was well able to handle all the animated models we have so far tested, and usually achieves a system utilization of at least 90%.

7.7.5 Remaining Limitations

Although we significantly reduce the dependence on refitting, we still refit for at least one frame. Thus, completely unstructured motion with near-randomly changing geometry every frame cannot be supported. However, practical applications for such completely random scenes are probably rare, and more likely effects, such as exploding objects, are arguably not worse than what happens in the BART and Exploding Dragon scenes, which our method handle well.

Similarly, relying on refitting does not allow for changing scene topology. However, this usually occurs only if entire objects appear into/disappear from a composite scene environment, which can easily be handled by a two-level approach as proposed in [77]; the small top-level scene could easily be rebuilt per frame, with our method handling the per-object deformations. In cases where completely new geometry is required—e.g., when a player enters a new level, or passes a portal—one could perform a single, parallel but synchronous rebuild. However, if known in advance, the rebuild could also be done asynchronously so that by the time it is required it will already be built. In fact, if rebuilds are fast enough to occur per frame, and we have found that they can be with 2–4 rebuild threads, then we could even handle completely dynamic scenes by rendering the BVH built during the previous frame, which is analogous to the double buffering used in rasterization. This is still advantageous since the overall frame rate is not the sum of the rebuild and render time, but just the max of the two.

7.8 Conclusion

In this chapter, we have presented a new approach to handling dynamic scenes in a highly parallel ray tracing system. Instead of trying to do a full BVH rebuild per frame, we rebuild asynchronously over the course of one or more frames, and in the meantime rely on refitting, which parallelizes well.

Our method's advantage over preexisting methods depends on the amount of deformation in a scene. If the deformation is sufficiently small, simply refitting every frame may suffice, rendering our method superfluous.

Since we still depend on a scene's deformability for at least short periods of time, we cannot handle randomly deforming scenes, or scenes with changing topology as ably as other data structures with faster complete rebuilds, such as a grid [80]. However, most scenes do not require full rebuilds each frame, and for those that do, we can still get up to twice the performance of previous BVH ray tracers by completely rebuilding the next frame's BVH while rendering the current frame.

Our methods are particularly designed for highly parallel multicore architectures. Although increasing parallelism is a problem for pure rebuilding, our methods in fact benefit from more cores, as the relative overhead decreases. As argued in the previous section, the currently foreseeable trends towards having many more cores, slightly more performance per core, and more rays per pixel would make our method even more suitable for these architectures than the one we used in our experiments.

Arguably the biggest limitation is that we have no way of predicting which configuration will work best for any given scene. Determining heuristics for doing that—potentially on the fly—would be an interesting avenue for future work.

CHAPTER 8

CONCLUSION

In this dissertation we have presented efficient acceleration structures for various situations. Unfortunately, there is no one best acceleration structure and so we must look into where each acceleration structure is best used.

8.1 Static Scenes

If preprocess time is not an issue, then the BSP is clearly the superior choice for most static scenes. For many scenes and camera views, the BSP is not significantly faster than other acceleration structures, which at first would make it seem not worth the very expensive build and increased code complexity. However, the BSP has the advantage that for certain scenes or viewpoints that cause other acceleration structures to slow down, the BSP can continue rendering at roughly the same high frame rate. This is a very important feature if consistently high frame rates are required. In fact, having a consistent frame rate is the same motivation for using an asynchronous BVH build over using the rebuild heuristic of Lauterbach et al. [52]. The situations that cause the axis-aligned structures to slow down can be partially fixed by splitting the problem triangles so that more empty space can be culled out; however, this is at the cost of increased traversals and memory usage. Comparing a kd-tree and BVH that allow for these splits with a BSP would be very interesting; especially so if the result is that all structures end up performing roughly as well, in which case the BSP, which is much more complicated to build, would not be recommended.

The coherent grid traversal is a fast way to trace packets of coherent rays through a grid. It is usually not as fast as frustum-based BSP, kd-tree, and BVH structures; but it can often be competitive and does allow for interactive frame rates. The coherent grid traversal, as well as single ray grid traversal, performs best when rays do not need to traverse many cells before hitting an object. A common situation where few grid traversals are required is when rendering the outside of an opaque object that fills most of the grid, such as a character model or the outside of a building. In fact, if it is unlikely that a ray will enter the model; then even if the model has a very complex interior, that cost will essentially be removed, unlike for tree-based structures for which the tree would become larger and more traversal steps required. Another situation where few traversal steps are required is when the scene is large and complex, but the camera is inside a small opaque object such that only a small subset of the scene is visible. An example of this would be a scene containing many buildings, each of which contain many rooms and the camera is inside a closed (opaque) room. A tree-based structure would have to traverse many levels before reaching the node that corresponded to the room; although that step can be made faster through an entry point (EP) traversal, it still is a cost that must occur for each ray packet [67]. The grid, on the other hand, can immediately start traversal with the cell that is in the room. Furthermore, if few grid traversals are required, then even very incoherent rays can be considered coherent with respect to grid traversal, making the use of a grid suitable for all types of rays. However, grid structures do not handle rooms very well since many traversal steps are required to traverse the room, especially if there are many objects scattered around the room which encumber multilevel grids from working as well. For this reason, it might be advantageous to render the room found using a coarse grid traversal with a tree-based structure.

For a general purpose acceleration structure with clearly defined coherent and incoherent ray packets that allow for easily choosing single ray traversal or packet-based traversal, then the kd-tree, or BSP if preprocess time can be ignored, are probably ideal for most static scenes. The grid or grid/BSP hybrid is probably ideal for the remaining static scenes. Ray casting and path tracing usually follow these well-defined conditions on ray coherency. If ray packet coherence is harder to determine, for instance in distribution ray tracing, then a BVH, which can gradually degrade to a single ray performance, might perform better. However, since this dissertation does not address how to better handle incoherent packets and instead relies on either using the same traversal algorithm in all cases, such as with the BVH, or having a user specified toggle between full packet tracing and single-ray tracing, it is quite likely that superior methods for handling incoherent rays might exist. Since we do not know what those methods might be, we used the single ray performance of incoherent rays and the packet performance of coherent rays to help guide our comparisons of the various acceleration structures with packets; however these comparisons could become invalidated with future research.

8.2 Dynamic Scenes

The choice of acceleration structure for dynamic scenes can very substantially based on what restrictions can be placed on the dynamic scene. For fully general scenes where we cannot make any assumptions, we must assume that full rebuilds are required. In addition, since teapot-in-a-stadium situations as well as incoherent ray packets could occur, it would be best to use a BVH or kd-tree with full parallel rebuilds. When teapot-in-a-stadium issues are known to not occur or can be avoided through multilevel grids, and the ray packets are fairly coherent, then the very fast rebuild of the coherent grid traversal may make it the superior acceleration structure.

When the render time is close to the rebuild time, then on a multicore system, up to a 2× speedup can be obtained by double buffering the acceleration structure and building the next frame's structure while rendering with the current frame. This also means that more expensive builds that do not take longer than the render time can be used, such as for a BVH or kd-tree, without the increased build time harming the overall frame rate.

If the type of dynamic scene can be restricted so that refitting can be used, then an asynchronous BVH would likely always be ideal since the BVH is very fast to render and the refit update time is very small. Even if it is not allowed, it might be worth trying to modify the scene so that it can be used. For instance, this could be done by moving nonexistent objects out of the scene or making them degenerate so they cannot be intersected. Most dynamic scenes can likely be made to fall into this category, making the asynchronous BVH a very useful acceleration structure.

If fast acceleration structure updates are not important, perhaps because only a few hierarchical transformations are required to animate the entire scene, then using another acceleration structure will likely give better results, with the BSP being most suitable if build time can be completely ignored.

8.3 Future Work

As the BSP showed, for certain types of scenes and camera positions, an improved acceleration structure can offer substantial performance improvements by fixing the shortfalls of preexisting structures. However, for well-behaved scenes where most of the triangles are visible and compact, for instance with laser scanned models, achieving substantially better performance is much more challenging for several reasons.

The first reason simply has to do with Amdahl's law. Since the total time to produce an image includes ray generation, shading, and writing the result to the frame buffer, even an infinite speedup in traversal and intersection time will only result in small overall speedups. Currently, the time spent outside of tracing and intersecting rays will often account for 20–50% of the total time, thereby limiting possible overall speedups to at most 5× even if the traversal and intersection time go completely away. Therefore, the only way to achieve further large speedups is to improve the performance of the entire ray tracing system.

The other main reason is that the current acceleration structures are already highly efficient for well-behaved scenes. For instance, in the conference room a kd-tree only performs about 4 triangle tests per ray and about 30 ray traversals. In order to get a 4× speedup in just tracing and testing rays, on average, the number of triangle tests and ray traversals would need to go down to about 1 and 8 per ray, respectively. Since a depth of 8 corresponds to only 256 leaves and the conference room has hundreds of thousands of triangles, it is highly unlikely that a kd-tree could achieve this simply by building a better tree. Likewise, obtaining the speedup from just optimizing the traversal step is also unlikely as traversal is also already very efficient. Both faster and fewer traversal steps and intersection tests would likely be required in order to get the speedup and even then, it would be extremely difficult if even possible.

The solution then is to improve the acceleration structure algorithms so that fewer traversals per ray are required. This can be done either by lowering the number of traversals required per ray or by allowing many rays to share the same traversal step. Arvo and Kirk's 5D structure [4] might be an example of the latter if we ignore the time to build the tree, and frustum traversal, where many rays share a single cheap traversal step, is a successful example of the former option. The challenge with frustum traversal and other packet-based techniques is allowing them to scale with more incoherent rays.

Adapting acceleration structures for future hardware is another important avenue for future work. SIMD, especially more than 4-wide, and GPUs are current challenges. Future hardware with fixed functions that aid acceleration structures could offer interesting opportunities.

In addition to improved acceleration structures that offer large improvements to certain types of scenes or for specialized hardware such as SIMD, there are still many opportunities for improved structures that further accelerate all static scenes on all types of hardware; however, order of magnitude speedups should not be expected. For dynamic scenes, there is likely still plenty of room for growth in improving the update to render time trade-off.

REFERENCES

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

[2] Amanatides, J., and Woo, A. A fast voxel traversal algorithm for ray tracing. *Eurographics '87* (1987), 3–10.

[3] Ar, S., Montag, G., and Tal, A. Deferred, self-organizing BSP trees. *Computer Graphics Forum 21*, 3 (2002), 269–278.

[4] Arvo, J., and Kirk, D. Fast ray tracing by ray classification. *Computer Graphics (SIGGRAPH '87 Proceedings) 21*, 4 (July 1987), 55–64.

[5] Arvo, J., and Kirk, D. A survey of ray tracing acceleration techniques. In *An Introduction to Ray Tracing*, A. S. Glassner, Ed. Academic Press, San Diego, CA, 1989, pp. 201–262.

[6] Bartz, D. Optimizing memory synchronization for the parallel construction of recursive tree hierarchies. In *Proceedings of Eurographics Workshop on Parallel Graphics and Visualization* (2000), pp. 53–60.

[7] Bartz, D., Grosso, R., Ertl, T., and Straer, W. Parallel construction and isosurface extraction of recursive tree structures. In *Proceedings of WSCG'98* (1998), vol. 3.

[8] Benthin, C. *Realtime Ray Tracing on Current CPU Architectures.* PhD thesis, Saarland University, 2006.

[9] Benthin, C., Wald, I., and Slusallek, P. A scalable approach to interactive global illumination. *Computer Graphics Forum 22*, 3 (2003), 621–630. (Proceedings of Eurographics).

[10] Boulos, S., Edwards, D., Lacewell, J. D., Kniss, J., Kautz, J., Shirley, P., and Wald, I. Packet-based whitted and distribution ray tracing. In *Proceedings of Graphics Interface 2007* (May 2007), pp. 177–184.

[11] Budge, B. C., Coming, D., Norpchen, D., and Joy, K. I. Accelerated building and ray tracing of restricted BSP trees. In *Proceedings of the 2008 IEEE/Eurographics Symposium on Interactive Ray Tracing* (2008), pp. 167–174.

[12] Butler, L., and Stephens, A. Bullet ray vision. In *Proceedings of the 2007 IEEE/Eurographics Symposium on Interactive Ray Tracing* (2007), pp. 167–170.

[13] Cazals, F., Drettakis, G., and Puech, C. Filtering, clustering and hierarchy construction: a new solution for ray tracing very complex environments. In *Proceedings of Eurographics '95* (1995), pp. 371–382.

[14] Christensen, P. H., Fong, J., Laur, D. M., and Batali, D. Ray tracing for the movie 'Cars'. In *Proc. IEEE Symposium on Interactive Ray Tracing* (2006), pp. 1–6.

[15] Cleary, J., Wyvill, B., Birtwistle, G., and Vatti, R. A parallel ray tracing computer. In *Proceedings of the Association of Simula Users Conference* (1983), pp. 77–80.

[16] Cleary, J. G., and Wyvill, G. Analysis of an algorithm for fast ray tracing using uniform space subdivision. *The Visual Computer 4*, 2 (1988), 65–83.

[17] Cohen, D., and Sheffer, Z. Proximity clouds–an acceleration technique for 3D grid traversal. *The Visual Computer 11*, 1 (1994), 27–38.

[18] Dammertz, H., and Keller, A. The edge volume heuristic—robust triangle subdivision for improved BVH performance. In *Proceedings of the 2008 IEEE/Eurographics Symposium on Interactive Ray Tracing* (2008), pp. 155–158.

[19] de Berg, M. Linear size binary space partitions for fat objects. In *ESA '95: Proceedings of the Third Annual European Symposium on Algorithms* (1995), Springer-Verlag London, UK, pp. 252–263.

[20] DeMarle, D. E., Gribble, C., and Parker, S. Memory-savvy distributed interactive ray tracing. In *Eurographics Symposium on Parallel Graphics and Visualization* (2004), pp. 93–100.

[21] Devillers, O. *Méthodes d'Optimisation du Tracé de Rayons*. PhD thesis, Université de Paris-sud, 1988.

[22] Devillers, O. The macro-regions: an efficient space subdivision structure for ray tracing. In *Proceedings of Eurographics '89* (September 1989), Elsevier Science Publishers, pp. 27–38.

[23] Djeu, P., Hunt, W., Wang, R., Elhassan, I., Stoll, G., and Mark, W. R. Razor: An architecture for dynamic multiresolution ray tracing. Tech. Rep. UTCS TR-07-52, University of Texas at Austin Dept. of Comp. Sciences, Jan. 2007. Conditionally accepted to ACM Transactions on Graphics.

[24] Dmitriev, K., Havran, V., and Seidel, H.-P. Faster ray tracing with SIMD shaft culling. Research Report MPI-I-2004-4-006, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 2004.

[25] Ernst, M., and Greiner, G. Early split clipping for bounding volume hierarchies. In *Proceedings of the 2007 IEEE/Eurographics Symposium on Interactive Ray Tracing* (2007), pp. 73–78.

[26] Farias, M., Scheidegger, C., Comba, J., and Velho, L. Boolean operations on surfel-bounded objects using constrained bsp-trees. In *Computer Graphics and Image Processing, 2005. SIBGRAPI 2005. 18th Brazilian Symposium on* (Oct. 2005), pp. 325–332.

[27] Fuchs, H., Kedem, Z. M., and Naylor, B. F. On visible surface generation by a priori tree structures. *SIGGRAPH Comput. Graph. 14*, 3 (1980), 124–133.

[28] Fujimoto, A., Tanaka, T., and Iwata, K. ARTS: Accelerated ray tracing system. *IEEE CG&A 6*, 4 (1986), 16–26.

[29] Glassner, A. S. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications 4*, 10 (1984), 15–22.

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

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

[32] Havran, V. Mailboxing, yea or nay? *Ray Tracing News 15*, 1 (2002).

[33] Havran, V., and Bittner, J. On improving kd-tree for ray shooting. In *Proceedings of WSCG* (2002), pp. 209–216.

[34] Havran, V., Kopal, T., Bittner, J., and Žára, J. Fast robust BSP tree traversal algorithm for ray tracing. *Journal of Graphics Tools 2*, 4 (1998), 15–23.

[35] Humphreys, G., Houston, M., Ng, R., Ahern, S., Frank, R., Kirchner, P., and Klosowski, J. T. Chromium: A stream processing framework for interactive graphics on clusters of workstations. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2002) 21*, 3 (July 2002), 693–702.

[36] Hunt, W., Mark, W. R., Fussell, D. S., and Stoll, G. Fast and lazy build of acceleration structures from scene hierarchies. In *Proceedings of the 2007 IEEE/EG Symposium on Interactive Ray Tracing* (2007), pp. 47–54.

[37] Hunt, W., Stoll, G., and Mark, W. Fast kd-tree construction with an adaptive error-bounded heuristic. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 81–88.

[38] Ize, T., Shirley, P., and Parker, S. G. Grid creation strategies for efficient ray tracing. In *Proceedings of the 2007 IEEE/Eurographics Symposium on Interactive Ray Tracing* (2007), pp. 27–32.

[39] Ize, T., Wald, I., and Parker, S. G. Asynchronous BVH construction for ray tracing dynamic scenes on parallel multi-core architectures. In *Proceedings of the 2007 Eurographics Symposium on Parallel Graphics and Visualization* (2007), pp. 101–108.

[40] Ize, T., Wald, I., and Parker, S. G. Ray tracing with the BSP tree. In *Proceedings of the 2008 IEEE/Eurographics Symposium on Interactive Ray Tracing* (2008), pp. 159–166.

[41] Ize, T., Wald, I., Robertson, C., and Parker, S. G. An evaluation of parallel grid construction for ray tracing dynamic scenes. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 47–55.

[42] Jansen, F. Data structures for ray tracing,. In *Proceedings of the Workshop in Data structures for Raster Graphics* (1986), pp. 57–73.

[43] Jevans, D., and Wyvill, B. Adaptive voxel subdivision for ray tracing. *Proceedings of Graphics Interface '89* (June 1989), 164–172.

[44] Kammaje, R. P., and Mora, B. A study of restricted BSP trees for ray tracing. In *Proceedings of the 2007 IEEE/Eurographics Symposium on Interactive Ray Tracing* (2007), pp. 55–62.

[45] Kay, T., and Kajiya, J. Ray tracing complex scenes. In *Proceedings of SIGGRAPH* (1986), pp. 269–278.

[46] Keates, M. J., and Hubbold, R. J. Interactive ray tracing on a virtual shared-memory parallel computer. *Computer Graphics Forum 14*, 4 (Oct. 1995), 189–202.

[47] Kensler, A., and Shirley, P. Optimizing ray-triangle intersection via automated search. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 33–38.

[48] Kirk, D., and Arvo, J. The ray tracing kernel. In *Proceedings of Ausgraph* (1988), pp. 75–82.

[49] Kirk, D., and Arvo, J. Improved ray tagging for voxel-based ray tracing. In *Graphics Gems II*, J. Arvo, Ed. Academic Press, 1991, pp. 264–266.

[50] Klimaszewski, K. S., and Sederberg, T. W. Faster ray tracing using adaptive grids. *IEEE CG&A 17*, 1 (Jan./Feb. 1997), 42–51.

[51] Lagae, A., and Dutré, P. Compact, fast and robust grids for ray tracing. *Computer Graphics Forum (Proceedings of the 19th Eurographics Symposium on Rendering) 27*, 4 (2008), 1235–1244.

[52] Lauterbach, C., Yoon, S.-E., Tuft, D., and Manocha, D. RT-DEFORM: Interactive ray tracing of dynamic scenes using bvhs. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 39–45.

[53] Lext, J., and Akenine-Möller, T. Towards rapid reconstruction for animated ray tracing. In *Eurographics Short Presentations* (2001), pp. 311–318.

[54] Lext, J., Assarsson, U., and Möller, T. BART: A benchmark for animated ray tracing. *IEEE Computer Graphics & Applications 21*, 2 (2001), 22–31.

[55] Ma, K.-L., Painter, J. S., Hansen, C. D., and Krogh, M. F. Parallel volume rendering using binary-swap compositing. *IEEE Comput. Graph. Appl. 14*, 4 (1994), 59–68.

[56] MacDonald, J. D., and Booth, K. S. Heuristics for ray tracing using space subdivision. *Visual Computer 6*, 6 (1990), 153–65.

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

[58] Möller, T., and Trumbore, B. Fast, minimum storage ray triangle intersection. *Journal of Graphics Tools 2*, 1 (1997), 21–28.

[59] Molnar, S., Cox, M., Ellsworth, D., and Fuchs, H. A sorting classification of parallel rendering. *IEEE Comput. Graph. Appl. 14*, 4 (1994), 23–32.

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

[61] Parker, S., Parker, M., Livnat, Y., Sloan, P.-P., Hansen, C., and Shirley, P. Interactive ray tracing for volume visualization. *IEEE Trans. on Computer Graphics and Visualization 5*, 3 (1999), 238–250.

[62] Parker, S. G., Martin, W., Sloan, P.-P. J., Shirley, P., Smits, B. E., and Hansen, C. D. Interactive ray tracing. In *Proceedings of Interactive 3D Graphics* (1999), pp. 119–126.

[63] Paterson, M. S., and Yao, F. F. Efficient binary space partitions for hidden-surface removal and solid modeling. *Discrete and Computational Geometry 5*, 1 (1990), 485–503.

[64] Popov, S., Günther, J., Seidel, H.-P., and Slusallek, P. Experiences with streaming construction of sah kd-trees. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 89–94.

[65] Reinhard, E., Smits, B., and Hansen, C. Dynamic acceleration structures for interactive ray tracing. In *Proceedings of the Eurographics Workshop on Rendering* (Brno, Czech Republic, June 2000), pp. 299–306.

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

[67] Reshetov, A., Soupikov, A., and Hurley, J. Multi-level ray tracing algorithm. *ACM Transaction on Graphics 24*, 3 (2005), 1176–1185. (Proceedings of ACM SIGGRAPH 2005).

[68] Rubin, S., and Whitted, T. A 3D representation for fast rendering of complex scenes. In *Proceedings of SIGGRAPH* (1980), pp. 110–116.

[69] Shevtsov, M., Soupikov, A., and Kapustin, A. Fast and scalable kd-tree construction for interactively ray tracing dynamic scenes. *Computer Graphics Forum 26*, 3 (2007). (Proceedings of Eurographics).

[70] Spackman, J. *Scene Decompositions for Accelerated Ray Tracing.* PhD thesis, The University of Bath, UK, 1990. Available as Bath Computer Science Technical Report 90/33.

[71] Spackman, J., and Willis, P. The SMART navigation of a ray through an oct-tree. *Computers and Graphics 15*, 2 (1991), 185–194.

[72] Sung, K., and Shirley, P. Ray tracing with the BSP tree. In *Graphics Gems III*, D. Kirk, Ed. Academic Press, San Diego, 1992.

[73] Takeuchi, A., Ino, F., and Hagihara, K. An improved binary-swap compositing for sort-last parallel rendering on distributed memory multiprocessors. *Parallel Comput. 29*, 11-12 (2003), 1745–1762.

[74] Wächter, C., and Keller, A. Instant ray tracing: The bounding interval hierarchy. In *Rendering Techniques 2006 – Proceedings of the 17th Eurographics Symposium on Rendering* (2006), pp. 139–149.

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

[76] Wald, I. On fast construction of SAH-based bounding volume hierarchies. In *Proceedings of the 2007 IEEE/Eurographics Symposium on Interactive Ray Tracing* (2007), pp. 33–40.

[77] Wald, I., Benthin, C., and Slusallek, P. Distributed interactive ray tracing of dynamic scenes. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics* (2003), pp. 11–20.

[78] Wald, I., Boulos, S., and Shirley, P. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics 26*, 1 (2007), 1–18.

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

[80] Wald, I., Ize, T., Kensler, A., Knoll, A., and Parker, S. G. Ray tracing animated scenes using coherent grid traversal. *ACM Transactions on Graphics 25*, 3 (2006), 485–493. (Proceedings of ACM SIGGRAPH).

[81] Wald, I., Ize, T., and Parker, S. G. Fast, parallel, and asynchronous construction of BVHs for ray tracing animated scenes. *Computers & Graphics 32*, 1 (2008), 3–13.

[82] Wald, I., Mark, W. R., Günther, J., Boulos, S., Ize, T., Hunt, W., Parker, S. G., and Shirley, P. State of the art in ray tracing animated scenes. In *State of the Art Reports, Eurographics 2007* (2007).

[83] Wald, I., Slusallek, P., and Benthin, C. Interactive distributed ray tracing of highly complex models. In *Rendering Techniques* (2001), S. J. Gortler and K. Myszkowski, Eds., Proceedings of the 12th Eurographics Workshop on Rendering Techniques, London, UK, June 25-27, 2001, Springer, pp. 274–285.

[84] Wald, I., Slusallek, P., Benthin, C., and Wagner, M. Interactive rendering with coherent ray tracing. *Computer Graphics Forum 20*, 3 (2001), 153–164. (Proceedings of Eurographics).

[85] Walter, B., and Shirley, P. Cost analysis of a monte-carlo radiosity algorithm. Tech. Rep. PCG-95-3, Program of Computer Graphics, Cornell University, 1995.

[86] Whitted, T. An improved illumination model for shaded display. *Communications of the ACM 23*, 6 (1980), 343–349.

[87] Woop, S., Schmittler, J., and Slusallek, P. RPU: A programmable ray processing unit for realtime ray tracing. *ACM Transactions on Graphics 24*, 3 (2005), 434–444. (Proceedings of SIGGRAPH).

[88] Yoon, S.-E., Curtis, S., and Manocha, D. Ray tracing dynamic scenes using selective restructuring. In *Eurographics Symposium on Rendering* (2007), pp. 73–84.